CHAPMAN AND HALL COMPUTING

Apple IIc and IIe Assembly Language

JULES H. GILDER

Apple IIc and IIe Assembly Language

Apple IIc and IIe Assembly Language

Jules H. Gilder

Chapman and Hall
New York London

First published 1986
by Chapman and Hall
29 West 35 Street, New York, N.Y. 10001

Published in Great Britain by
Chapman and Hall Ltd
11 New Fetter Lane, London EC4P 4EE

ISBN-13: 978-0-412-01121-4 e-ISBN-13: 978-1-4684-6424-5
DOI: 10.1007/978-1-4684-6424-5

Library of Congress Cataloging-in-Publication Data

Gilder, Jules H., 1947-
Apple IIc and IIe assembly language.

Bibliography: p.
1. Apple IIc (Computer)--Programming.
2. Apple IIe (Computer--Programming. 3. Assembler language (Computer program language) I. Title.
II. Title: Apple 2c and 2e assembly language.
III. Title: Apple two c and 2 e assembly language.
QA76.8.A66225G55 1986 005.265 86-2592

DEDICATION

Dedicated with love to Miriam.

Contents

PREFACE

The Apple // series of computers represents one of the most versatile and powerful home computers available. If you've used your computer for a while, you've probably become quite familiar with Applesoft BASIC. That's good, because once you know that, this book will show you how to graduate from BASIC programming to assembly language programming. There are many reasons to program your Apple in assembly language. First and foremost is speed. Assembly language is about 100 times faster than BASIC. If you're thinking of writing games or business programs that do sorting, speed is of the essence and assembly language is a must. Assembly language programs usually also require less memory. Thus you can squeeze more complex programs into a smaller amount of memory. Finally, assembly language programs offer you a considerable amount of security, because they are more difficult to trace and change.

While assembly language is powerful, it doesn't have to be difficult to learn. In fact, if you can write programs in Applesoft BASIC, you're already half-way home. This book assumes you know BASIC and absolutely nothing about assembly language or machine language. Every effort has been made to write in nontechnical language and to set the chapters out in a logical manner, introducing new concepts in digestible pieces as and when they are needed, rather than devoting whole chapters to specific items. Wherever possible, practical programs are included to bring home the point being made, and in most instances they are analyzed and the function and operation of each instruction explained.

Apple //c and //e Assembly Language is completely self-contained, includes a full description of all the machine-code instructions available, and suggests suitable applications for their use. After a bit of theory in the opening chapters, the main registers of the 6502 and 65C02 are introduced and descriptions given of how, when, and where machine code routines can be entered. There is also a simple machine code monitor program to facilitate the entry of such routines.

After discussing the ways in which the microprocessor flags certain conditions to the outside world, some of the modes of addressing the chip are described. Machine code addition and subtraction routines are introduced, and the easiest ways of manipulating and saving data for future use by the program and the processor are described. Machine code loops, (equivalent to BASIC's FOR...NEXT...STEP) show how sections of code may be repeated, and subroutines and jumps take the place of BASIC's GOTO and GOSUB. Also included is a look at some of the more complicated procedures, such as multiplication and division, using the shift and rotate instructions.

Details of the routines in the Apple's ROMs are provided to make your programming task easier. And, finally, a comprehensive set of appendices provides quick reference to the sort of things you'll need to "want to know quickly" when you start writing your very own original machine code programs.

You've just taken your first step into an exciting, new world of computing. Good luck and happy computing.

Jules H. Gilder

Apple IIc and IIe Assembly Language

1 Machine Code or Assembly Language

The 6502 (65C02) microprocessor in your Apple // computer can perform 151 (178) different operations, with each one being defined by a number (or operation code) in the range 0 to 255. To create a machine code program, we need simply to POKE successive memory locations with the appropriate operation codes—"opcodes" for short. For example, to store the value 5 at location 1500 (in other words, to do the machine code equivalent of Applesoft BASIC's POKE 1500,5), we would need to POKE the following bytes into memory:

```
169
  5
141
220
  5
```

and then ask the Apple's microprocessor to execute them. Not exactly clear, is it? That's where assembly language comes in. Assembly language allows us to write machine code in an abbreviated, form which is designed to represent the actual operation the opcode will perform. That abbreviated form is known as mnemonic, and it is the basic building block of assembly language (or assembler) programs.

We could rewrite the previous machine code in assembler like this:

```
LDA #5
STA 1500
```

and it can be read as:

Load the **a**ccumulator with the value 5.
Store the **a**ccumulator's content at location 1500.

As you can see from the **bold** letters, the mnemonic is composed of letters in the instruction, which greatly enhances its readability.

Once the assembler program is complete, it can be converted into machine code in one of two ways.

1. With the aid of a mnemonic assembler. That is itself a program (written in machine code or BASIC), which transforms the assembly language instructions (known as the source) into machine code (known as the object code) and POKEs them into the memory as it does so.

2. By looking up the relative codes in a table and then POKEing them into memory using a monitor program or a DATA- reading FOR...NEXT loop. Full details of that method are given in Chapter 6, which also includes a simple monitor program.

All the programs in this book are listed in their DATA statement, machine code, and assembler forms, so they can be entered by any of the above methods. Simply extract the information you require.Appendix 3 provides comprehensive information about all of the 6502's (and 65C02's) opcodes, so don't worry if some of this seems a bit foreign at the moment—we'll soon change that!

WHY MACHINE CODE?

A question often asked is "Why bother to program in machine code at all?" Well, one reason might be that you're fed up with BASIC and want to broaden your horizons, but, from the practical point of view there are two main reasons for programming in machine code.

First, speed. Machine code is executed very much faster than an interpreted high-level language, such as Applesoft BASIC. Remember that the BASIC interpreter is itself written in machine code, and that the BASIC statements and commands are simply pointers to the machine code routines in the ROM, which actually carry out the specified functions. It is because each statement and command must first be identified and located within the ROM that a decrease in operational speed occurs. Second, learning machine code allows you to understand just how your computer works and lets you create special effects and routines not possible within the constraints imposed by the limited set of BASIC instructions. Machine code allows you to control your Apple // rather than letting it controlling you!

2 Numbers

BINARY, HEX, AND DECIMAL

We have seen that the instructions the Apple // operates with consist of sequences of numbers. But just how are these numbers stored internally? Well, not wishing to baffle you with the wonders of modern computer science, let's try to simplify matters somewhat and say that each instruction is stored internally as a binary number. Decimal numbers are composed of combinations of 10 different digits, that is, 0,1,2,3,4, 5,6,7,8, and 9 and are said to work to a *base* of 10. As the name suggests, binary numbers represent the two different electrical conditions that are available inside the Apple // computer, namely, 0 volts (off) and 5 volts (on).

The machine code described in Chapter 1 is therefore represented internally as:

Mnemonic	Machine code	Binary
LDA	169	10101001
$5	05	00000101
STA	141	10001101
00	00	00000000
$15	15	00010101

As can be seen, each machine code instruction is expressed as 8 binary digits, called *bits*, which are collectively termed a *byte*. Usually each of the bits in a byte is numbered for convenience as follows:

7	6	5	4	3	2	1	0

The number of the bit increases from right to left, but that is not so odd as it may first seem.

Consider the decimal number 2934. We read it as two thousand, nine hundred, and thirty-four. The highest numerical value, two thousand, is on the left, while the lowest, four, is on the right. We can thus see from that the position of the digit in the number is very important, as it will affect its *weight*.

The second row of Table 2.1 introduces a new numerical representation. Each base value is postfixed with a small number or power, which corresponds to its overall position in the number. Thus 10^3, read as 10 raised to the *power* of 3, simply implies 10 x 10 x 10 = 1000.

Table 2.1

Value	1000s	100s	10s	1s
Representation	10^3	10^2	10^1	10^0
Digit	2	9	3	4

In binary representation the weight of each bit is calculated by raising the base value, 2, to the bit position (see Table 2.2). For example, bit number 7 has a notational representation of 2^7, which expands to 2 x 2 x 2 x 2 x 2 x 2 x 2 = 128.

Table 2.2

Bit number	7	6	5	4	3	2	1	0
Representation	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
Weight	128	64	32	16	8	4	2	1

BINARY TO DECIMAL CONVERSION

As it is possible to calculate the weight of individual bits, it is a simple matter to convert binary numbers into decimal numbers. The rules for conversion are as follows:

1. If the bit is *set*—that is, it contains a 1—add its weight.
2. If the bit is *clear*—that is, it contains a 0—ignore its weight.

Let us try an example and convert the binary number 10101010 into its equivalent decimal value.

$$
\begin{aligned}
1 \times 128(2^7) &= 128 \\
0 \times 64(2^6) &= 0 \\
1 \times 32(2^5) &= 32 \\
0 \times 16(2^4) &= 0 \\
1 \times 8(2^3) &= 8 \\
0 \times 4(2^2) &= 0 \\
1 \times 2(2^1) &= 2 \\
0 \times 1(2^0) &= 0 \\
\hline
&170
\end{aligned}
$$

Therefore, 10101010 binary is 170 decimal.

Similarly, 11101110 represents:

$$
\begin{aligned}
1 \times 128(2^7) &= 128 \\
1 \times 64(2^6) &= 64 \\
1 \times 32(2^5) &= 32 \\
0 \times 16(2^4) &= 0 \\
1 \times 8(2^3) &= 8 \\
1 \times 4(2^2) &= 4 \\
1 \times 2(2^1) &= 2 \\
0 \times 1(2^0) &= 0 \\
&\hline \\
&238
\end{aligned}
$$

in decimal.

DECIMAL TO BINARY CONVERSION

To convert a decimal number into a binary one, the procedure described earlier is reversed—each binary weight is subtracted in turn, starting with the highest binary weight, which is 128 for the standard 8-bit byte. If the subtraction is possible, a 1 is placed into the binary column and the remainder is carried down to the next row, where the next lowest binary weight is subtracted.

If the subtraction is not possible, a 0 is placed in the binary column and the number is moved down to the next row. For example, the decimal number 141 is converted into binary as in Table 2.3.

Table 2.3

Decimal number	Binary weight	Binary	Remainder
141	$128(2^7)$	1	13
13	$64(2^6)$	0	13
13	$32(2^5)$	0	13
13	$16(2^4)$	0	13
13	$8(2^3)$	1	5
5	$4(2^2)$	1	1
1	$2(2^1)$	0	1
1	$1(2^0)$	1	0

The must significant digit of the resulting binary number (the leftmost digit) is at the top of the column with progressively less significant digits beneath it. Thus, as can be seen from Table 2.3, the number 141 in decimal is equal to 100001101 in binary.

BINARY TO HEX CONVERSION

Although binary notation is probably as close as we can come to representing the way numbers are stored within the Apple // computer, you will no doubt have noticed that the machine code examples include some groups of characters preceded by a dollar sign ($). They represent another type of number, known as a hexadecimal number or hex for short. Hexadecimal numbers are calculated to a base of 16 instead of 10 as in decimal, or 2, as in binary. That at first might seem quite strange; however, it does present several distinct advantages over binary and decimal numbers, as we shall see.

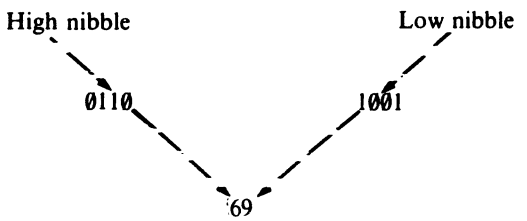
Sixteen different characters are required to represent all the possible digits in a hex number. To produce them, the numbers 0 through 9 are retained, and the letters A, B, C, D, E, and F are use to denote the values 10 to 15. Binary conversion are shown in Table 2.4

Table 2.4

Decimal	Hex	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111

To convert a binary number into hex, the byte must be separated into two sets of four bits, termed *nibbles*, and the corresponding hex value of each nibble extracted from Table 2.4.

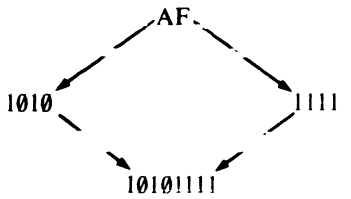
Example Convert 0110 1001 to hex:



Because it is not always apparent whether a number is hex or decimal (as in the example above), hex numbers on the Apple are always preceded by a dollar sign- therefore 01101001 is $69 (read hex six nine).

By reversing the process, hex numbers can readily be converted into binary.

Example convert $AF to binary:



It should now be apparent that hex numbers are much easier to convert to binary (and vice versa), than their decimal counterparts, and that the maximum binary number possible with one byte, 11111111, requires just two hex digits, $FF.

HEX TO DECIMAL CONVERSION

For the sake of completeness, let's see how hex and decimal numbers can be converted. To transform a hex number into decimal, the decimal weight of each digit should be summed.

Example convert $31A to decimal:

The 3 has the value $3 \times 16^2 = 3 \times 16 \times 16 = 768$

The 1 has the value $1 \times 16^1 = 1 \times 16 = 16$

The A has the value $10 \times 16^0 = 10 \times 1 = 10$

add these together to give $31A = 794 decimal.

Converting decimal to hex is a bit more involved and requires the number to be repeatedly divided by 16 until a value less than 16 is obtained. This hex value is noted, and the remainder carried forward for further division. This process is continued until the remainder itself is less than 16.

Example convert 4072 to hex:

$4072 \div 16 \div 16 = 15 = F$ (remainder = $4072 - (15 \times 16 \times 16) = 232$)

$232 \div 16 = 14 = E$ (remainder = $232 - (14 \times 16) = 8$)

$8 = 8$

Therefore 4072 decimal is $FE8.

Both of these conversions are a little long winded (to say the least!) and after all we do have a very sophisticated microcomputer available to us, so let's make it do some of this more tedious work!

3 It All Adds Up!

BINARY ARITHMETIC

Please don't be put off and skip this chapter simply because it contains that dreaded word—arithmetic. The addition and subtraction of binary numbers is simple, in fact if you can count to two you will have no problem whatsoever! Although it is not vital to be able to add and subtract ones and noughts by 'hand', this chapter will introduce several new concepts which are important, and will help you in your understanding in the next few chapters.

ADDITION

There are just four, simple, straightforward rules when it comes to adding binary numbers. They are:

1. 0+0=0
2. 1+0=1
3. 0+1=1
4. 1+1=(1)0

Note, that in rule four the result of 1+1 is (1)0. The 1 in the brackets is called a *carry* bit, and its function is to donate an overflow from one column to another, remember, 10 binary is 2 decimal. The binary 'carry' bit is quite similar to the carry that can occur when adding two decimal numbers together whose result is greater than 9. For example, adding together 9+1 we obtain a result of 10 (ten). That was obtained by placing a zero in the units column and carrying the 'overflow' across the next column to give: 9+1=10. Similarly, in binary addition when the result is greater than 1, we take the carry bit across to add to the next column.

Let's try to apply these principles to add together two 4-bit binary numbers, 0101 and 0100.

```
  0101        ($5)
+ 0100        ($4)
------
  1001        ($9)
```

Reading each individual column from right to left:

First column:	1 + 0	= 1
Second column:	0 + 0	= 0
Third column:	1 + 1	= 0 (1)
Fourth column:	0 + 0 = 0 + (1)	= 1

In this example a carry bit was generated in the third column, and was carried across and added to the fourth column.

Adding 8 bit numbers is accomplished in a similar manner:

01010101	($55)
+ 01110010	($72)
11000111	($C7)

SUBTRACTION

So far we have been dealing with positive numbers, however in the subtraction of binary numbers we need to be able to represent negative numbers as well as positive ones. In binary subtraction though, a slightly different technique from normal everyday subtraction is used. In fact we don't really perform a subtraction at all, we add the negative value of the number to be subtracted. For example, instead of executing 4-3 (four minus three) we actually execute 4+(-3) (four, plus minus three)! Figure 3.1 will hopefully eradicate any confusion or headaches that may be prevailing!

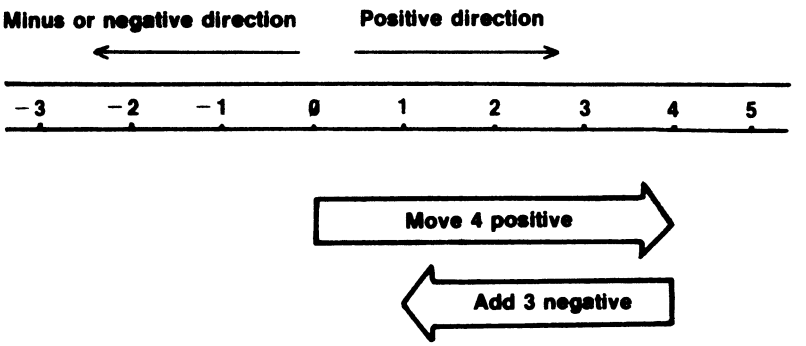
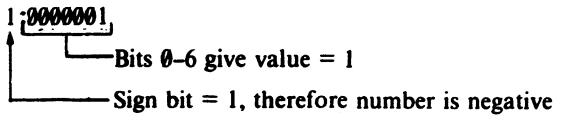


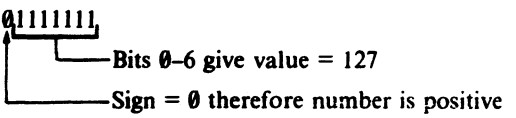
Figure 3.1 Diagramatic representation of 4 + (− 3).

We can use the scale to perform the example 4+(-3). The starting point is zero. First move to point 4 (i.e. four points in a positive direction).and add to this-3 (i.e. move three points in the negative direction). We are now positioned at point 1 which is, of course where we should be. Try using this method to subtract 8 from 12, to get the principle clear in your mind.

Okay, lets now see how we apply this to binary numbers, but first, just how are negative numbers represented in a binary? Well, a system known as signed binary is employed, where bit 7, known as the most significant bit (msb), is used to denote the sign of the number. Traditionally a '0' in bit 7 denotes a positive number and a '1' a negative number. For instance, in signed binary:



so, 10000001 = −1. And:



therefore 01111111=127.

However, just adjusting the value of bit 7 as required, is not an accurate way of representing negative numbers. What we must do to convert a negative number into its counterpart, is to obtain its *two's complement* value. To do this simply invert each bit and then add one.

00000011

Now invert each bit.(Replace each 0 with a 1, and each 1 with a 0—this is known as its *one's complement.*

11111100

Now add 1:

```
  1 1 1 1 1 1 0 0
+               1
-----------------
  1 1 1 1 1 1 0 1
```

Thus, the two's complement value of -3=11111101. Let us now apply this to our original sum 4+(-3):

```
(4)              0 0 0 0 0 1 0 0
(−3)             1 1 1 1 1 1 0 1
                 ---------------
(Now add)    (1)0 0 0 0 0 0 0 1
```

We can see that the result is 1 as we would expect, but we have also generated a carry bit due to an overflow from bit 7. This carry bit can be ignored for our purposes at present, though it does have a certain importance as we shall see later on.

A further example may be of use. Perform 32-16 i.e. 32+(- 16).

32 in binary is:

00100000

16 in binary is:

00010000

The two's complement of 16 is:

```
  11101111
+        1
----------
  11110000
```

Now we add the two together:

```
(32)        00100000
(-16)     + 11110000
(16)    (1) 00010000
```

Ignoring the carry, we have our result, 16.

We can see from these examples that, using the rules of binary addition, it is possible to add or subtract signed numbers. If the 'carry' is igna correct negative result. Using two's complement signed binary let's perform (-2)+(-2).

2 in binary is:

00000010

The two's complement value is:

```
  11111101
+        1
----------
  11111110
```

We can add this value twice to perform the addition:

```
(-2)        11111110
(-2)      + 11111110
        (1) 11111100
```

Ignoring the carry, the final result is -4. You might like to confirm this by obtaining the two's complement value of -4 in the usual manner.

BINARY CODED DECIMAL (BCD)

So far we have been dealing with the binary representation of hexadecimal numbers, which is the normal way the Apple // deals with its instructions and numbers. However, on certain occasions, such as when dealing with business applications, where it is essential to retain every significant digit in a result, it would be advantageous to work in a form of decimal binary where only the decimal digits 0 and 9 are available. Binary Coded Decimal, or BCD for short, allows us to do this. Table 3.1 shows the BCD-binary representations. As can be seen, only the binary values 0000 through to 1001 are required, and the combinations 1010 to 1111 are unused and are not legal values We can use the scale to perform the example 4+(-3). The starting point is zero. First move to point 4 (i.e. four points in a posin BCD.

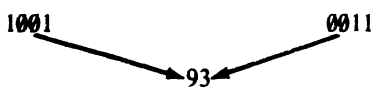
Table 3.1

BCD digit	Binary
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
not used	1010
not used	1011
not used	1100
not used	1101
not used	1110
not used	1111

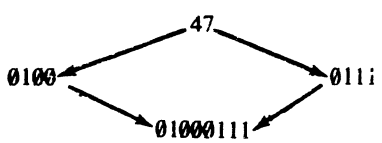
As only four binary bits are required to code any BCD digit, two BCD digits can be included in a single byte if required, this is known as *packed BCD.*

Converting BCD to binary and vice versa is performed as described in the previous hexadecimal examples; split the byte in half and convert each nibble separately.

Example convert 10010011 to BCD:



Example convert 47 BCD to binary:



We shall see how the Apple distinguishes between BCD and normal hex binary in Chapter 7, but first some sums!

BCD ADDITION

We can now try adding two BCD binary value, consider 8 BCD + 4 BCD:

8 BCD	0000 1000
4 BCD	0000 0100
Adding together	0000 1100

We have obtained an illegal result, in fact we have obtained the 'binary' sum and not the 'BCD' sum. The result should of course be 12 BCD which is 0001 0010 in BCD binary. In order to reach the correct value, the redundant binary values (1010 to 1111) must be 'jumped over' or adjusted. To do this 6(or 0110 binary) must be added. Thus,

0000 1100	(binary result)
0000 0110	(BCD correction)
0001 0010	(result 12 BCD)

We can try another example which includes this decimal adjustment – 15 BCD + 10 BCD:

15 BCD	0001 0101	
10 BCD	0001 0000	
Binary result	0010 0101	= 25 BCD

In this example the correct result has been obtained without adding the decimal adjust. This is because the addition was such that it did not encounter any of the illegal BCD values. This means that when adding two BCD digits, the decimal adjust value need only be added if a nibble value greater than 9 occurs in the result.

BCD SUBTRACTION

BCD subtraction is not so much difficult as it is involved. It is performed along the lines of binary subtraction, but instead of adding the two's complement value of the negative number, you add its *ten's complement*. The best way to explain the process is by working through an example. Let's choose a simple subtraction first of all, 9 BCD - 4 BCD:

9 BCD	0000 1001
4 BCD	0000 0100

To find the one's complement of 4 BCD invert each bit:

1111 1011

and add one to obtain the ten's complement (and thus -4 BCD):

```
 1111 1011
         1
-----------
 1111 1100
```

Perform the subtraction by adding the two values:

```
 9 BCD        0000 1001
-4 BCD        1111 1100
           (1)0000 0101
```

Ignore the carry to obtain the result: 5 BCD.

Simple so far, but what if we encounter something like 11 BCD - 5 BCD? Here adjustment will be necessary to take into account the six unused binary codes 1010 to 1111. Instead of adding the decimal adjustment, it must be subtracted.

```
11 BCD        0001 0001
-5 BCD        1111 1011
           (1)0000 1100
```

Ignoring the carry, we need to convert the illegal code to BCD by subtracting 6(or rather adding its two's complement).

```
Binary result      0000 1100
-6                 1111 1010
                (1)0000 0110
```

Result = 6 BCD, which is correct.

4 It's Logical

LOGICAL OPERATIONS

The theory of logic is based on situations where there can only be two possibilities, namely *yes* and *no*. In binary terms these two possibilities are represented as 1 and 0.

There are three different logical operations that can be performed on binary numbers, They are AND, OR and EOR. In each case the logical operation is performed between the corresponding bits of two separate numbers.

AND

The four rules for AND are:

1. 0 AND 0 = 0
2. 1 AND 0 = 0
3. 0 AND 1 = 0
4. 1 AND 1 = 1

As can clearly be seen, the AND operations will only generate a 1 if *both* of the corresponding bits being tested are 1. If a 0 exists in either of the corresponding bits being tested, the resulting bit will always be 0.

Example AND the following two binary numbers:

```
      1010
AND   0011
      ----
      0010
```

In the result only bit 1 is set, the other bits are all clear because in each case one of the bits being tested contains a 0.

The main use of the AND operation is to 'mask' or 'preserve' certain bits. Imagine that we wish to preserve the low four bits of a byte (low nibble) and completely clear the high four bits (high nibble). We would need to AND the number with 00001111. If the other byte contained 10101100 the result would be given by:

```
      10101100      (byte being tested)
AND   00001111      (mask)
      ----------
      00001100
```

the high nibble is cleared and the low nibble preserved!

OR

The four rules for OR are:

1. 0 OR 0 = 0
2. 1 OR 0 = 1
3. 0 OR 1 = 1
4. 1 OR 1 = 1

Here the OR operation will result in a 1 if *either* or *both* the bits contain a 1. A 0 will only occur if neither of the bits contains a 1.

Example OR the following two binary numbers:

```
      1010
OR    0011
      ----
      1011
```

Here, only bit 2 is clear, the other bits are all set as each pair of tested bits contains at least one 1.

One common use of the OR operation is to ensure that a certain bit (or bits) is set-this is sometimes called 'forcing bits'. As an example, if you wish to force bit 0 and bit 7, you would need to OR the other byte with 10000001.

```
      00110110      (byte being tested)
OR    10000001      (forcing byte)
      ----------
      10110111
```

The initial bits are preserved, but bit 0 and bit 7 are 'forced' to 1.

EOR

Like AND and OR, this donkey sounding operation has four rules:

1. 0 EOR 0 = 0
2. 1 EOR 0 = 1
3. 0 EOR 1 = 1
4. 1 EOR 1 = 0

This operation is exclusive to OR in other words, if both bits being tested are similar a 0 will result. A 1 will only be generated if the corresponding bits are unlike.

Example EOR the following two binary numbers:

```
        1010
EOR     0011
        ----
        1001
```

This instruction is often used to complement, or invert, a number. Do this by EORing the other byte with 11111111.

```
        10011000        (byte being inverted)
EOR     11111111        (inverting byte)
        --------
        01100111
```

Compare the result with the first byte, it is completely opposite.

5 The Registers

To enable the 6502 (65C02) to carry out its various operations, it contains within it several special locations, called *registers.* Because these registers are internal to the 6502 (65C02), they do not appear as part of the Apple //'s memory map, and are therefore referred to by name only. Figure 5.1 shows the typical programming model of the 6502 (65C02). For the time being we need only concern ourselves with the first four of these six registers, they are the accumulator, the X and Y registers and the Program Counter.

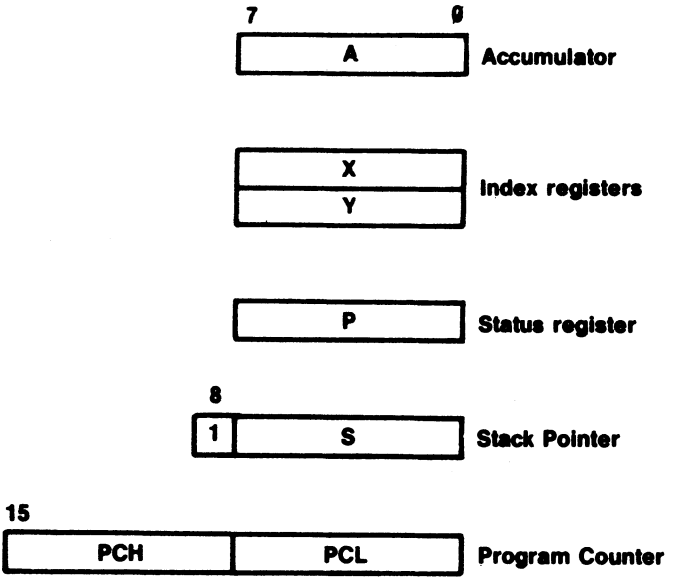


Figure 5.1 The registers—a typical programming model.

THE ACCUMULATOR

We have already mentioned the accumulator (or 'A' register) several times in the opening chapter. As you may have already gathered, the accumulator is the main register of the 6502 (65C02), and like most of the other registers it is eight bits wide. This means

that it can hold a single byte of information at any one time. Being the main register, it has the most instructions associated with it, and its principle feature is that all arithmetic and logical operations are carried out through it.

The accumulator's associated instructions are listed in Table 5.1. It is not absolutely vital to be familiar with these at present, but they are included now as an introduction.

Table 5.1

Accumulator instructions			
ADC	Add with carry	PHA	Push accumulator
AND	Logical AND	PLA	Pull accumulator
ASL	Arithmetic shift left	ROL	Rotate left
BIT	Compare memory bits	ROR	Rotate right
CMP	Compare to accumulator	SBC	Subtract with carry
EOR	Logical EOR	STA	Store the accumulator
LDA	Load the accumulator	TAX	Transfer accumulator to X register
LSR	Logical shift right	TAY	Transfer accumulator to Y register
ORA	Logical OR	TXA	Transfer X register to accumulator
		TYA	Transfer Y register to accumulator

THE INDEX REGISTERS

There are two further registers in the 6502 which can hold single byte data. These are the *X register* and the *Y register*. They are generally termed the 'index registers', because they are very often used to provide an 'offset' or index from a specified base address. They are provided with direct increment and decrement instructions to transfer the contents of these registers into the accumulator and vice versa.

The instructions associated with both registers are given in Table 5.2.

Table 5.2

X register instructions		Y register instructions	
CPX	Compare X register	CPY	Compare Y register
DEX	Decrement X register	DEY	Decrement Y register
INX	Increment X register	INY	Increment Y register
LDX	Load the X register	LDY	Load the Y register
PHX	Push X register onto stack	PHY	Push Y register onto stack
PLX	Pull top of stack into X register	PLY	Pull stack into Y register
STX	Store the X register	STY	Store the Y register
TAX	Transfer accumulator to X reg.	TAY	Transfer accumulator to Y reg.
TXA	Transfer X reg. to accumulator	TYA	Transfer Y reg. to accumulator
TSX	Transfer Status to X register		
TXS	Transfer X register to Status		

THE PROGRAM COUNTER

The Program Counter is the 6502's address book. It nearly always contains the address in memory where the next instruction to be executed sits. Unlike the other registers, it is a 16 bit register, consisting physically of two 8 bit registers. These two are generally referred to as Program Counter High (PCH) and Program Counter Low (PCL).

6 A Poke at Machine Code

Now that we have gotten some of the basics out of the way, why don't we write our first machine code program, after all, that's what this book is all about!

Enter Program 1-you can omit the REM statements if you like. The (hex) machine code and assembler versions of the instructions are included as REMs alongside the DATA statements (which are, of course, in decimal).

Program 1

```
10  REM * * MACHINE CODE DEMO * *
20  REM * PRINT 'A' ON SCREEN *
30  CODE = 768
40  FOR LOOP = 0 TO 5
50    READ BYTE
60    POKE CODE + LOOP, BYTE
70  NEXT LOOP
80  :
90  REM * * MACHINE CODE DATA * *
100 DATA 169,193    : REM $A9, $C1       — LDA #ASC"A"
110 DATA 32,237,253 : REM $20, $ED, $FD  — JSR 65005
120 DATA 96         : REM $60            — RTS
130 HOME
140 REM * * EXECUTE MACHINE CODE * *
150 CALL 768
```

The functions of this short program is to print the letter 'A' on the screen. Nothing spectacular, but the program does incorporate various features that will be common to all your future machine code programs. The meaning of each line is as follows:

Line 30 Declare a variable called CODE to denote where the machine code is placed.
Line 40 Set up a data-reading loop.
Line 50 Read one byte of the machine code data.

Line 60 POKE byte value into memory.
Line 70 Repeat loop until finished.
Line 100 Machine code data—place ASCII code for A in accumulator.
Line 110 Machine code data—print A on screen.
Line 120 Machine code data—Return to BASIC.
Line 130 Applesoft command to clear the screen.
Line 150 Execute the machine code

To see the effect of the program just type in RUN, hit the RETURN key and voila—the 'A' should be sitting at the top of the screen.

CODE-THE PROGRAM COUNTER

It should be fairly obvious that the machine code we write has to be stored somewhere in memory. In all the programs in this book I have used the BASIC variable 'CODE' as a pointer to the start address of the memory where the machine code is to be placed. (CODE acts, in effect, rather like the the processor's own Program Counter.) You may wish to use your own variable name—and this is perfectly acceptable. For example, you may consider that PC is a more appropriate name for the start of the code—or even MACHINECODE. It does not really matter. What does matter is that you should get into the habit of using the same variable name in all your programs, and thus avoid ambiguity.

The value given to CODE must be chosen with care. It would be easy enough to allocate an address which causes the machine code to overwrite another program or even the assembler program itself! In program 1 CODE is set to 768 using the normal variable assignment statement:

```
CODE = 768
```

and the six bytes of the machine code are stored there—or more correctly—in the six bytes starting at 768 (768 to 773). If you look at Figure 6.1 you will notice that this area is in what is known as page 3 of memory. Each page of memory contains 256 bytes, but not all 256 on page 3 are available for use. The Apple requires 48 of them, so it is only safe to use 208 of them from $300 to $3CF (768 to 975).

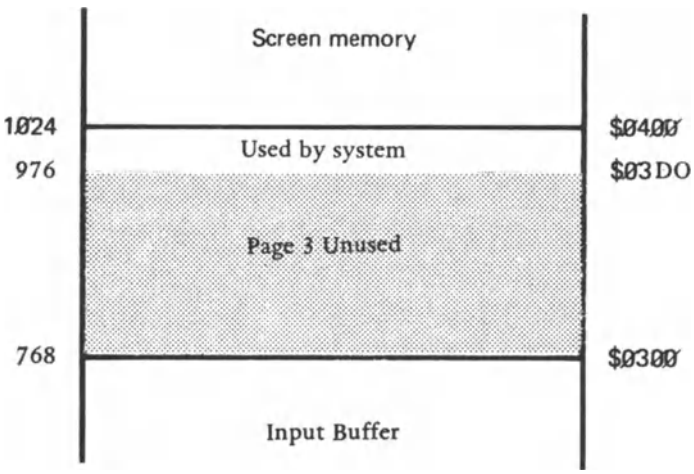


Figure 6.1 Place machine code in page 3 of memory.

BASIC programs are stored in the user RAM which stretches from 2048 ($800) to 38399 ($95FF) – a massive 35.5K. It is quite feasible to place your machine code programs here – but you must avoid conflict with Applesoft BASIC. If your program is entirely machine code then no problems should occur, however, if it is used in conjunction with Applesoft, then it must be assembled well out of harm's way. Perhaps the best area is at the top of available user RAM starting at 38399 ($95FF) and working down in memory. If, for example, you needed space for about 500 bytes of machine code, you could include the HIMEM: 37888 statement at the beginning of your Applesoft program and 2 pages of memory (512 bytes) would be reserved. The ProDOS operating system requires memory to be reserved in full pages, while in the DOS 3.3 system any amount of memory can be reserved with the HIMEM: statement (see Figure 6.2).

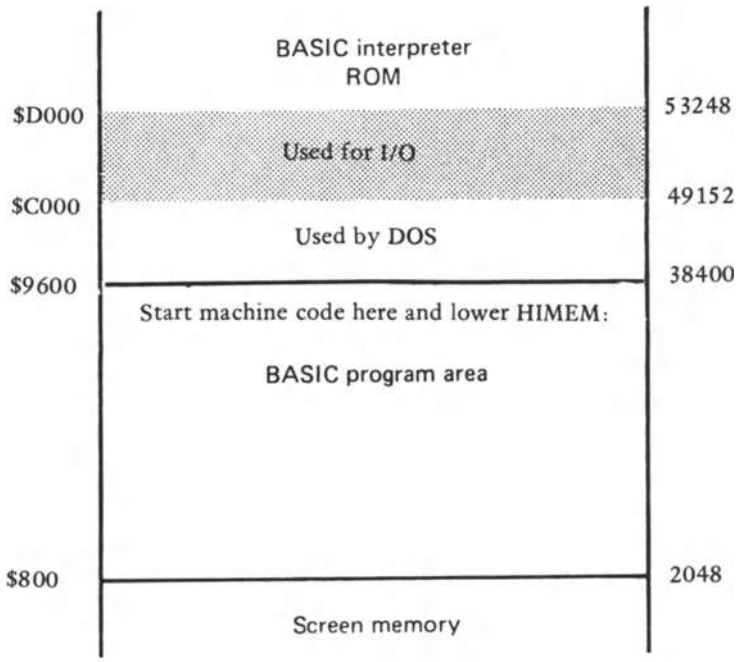


Figure 6.2 Place machine code in high RAM to prevent destruction by Applesoft.

By using the Applesoft's HIMEM: command, we are telling the Apple not to store its string variables at the end of DOS as it normally does, but to make a note to itself that the highest location of user available RAM has been changed and is now two pages lower in memory. In other words we're telling Applesoft that it cannot use any memory location above 37888. Program 2 illustrates how this technique can be used.

Program 2

```
 10  REM ** PLACE M/C BELOW DOS **
 20  REM ** RESET HIMEM TO 37888 **
 30  REM ** WHICH IS $9400 **
 40  HIMEM: 37888
 50  CODE = 37888
 80  FOR LOOP = 0 to 5
 90      READ BYTE
100      POKE CODE + LOOP,BYTE
```

```
110 NEXT LOOP
120 :
130 REM ** M/C DATA **
140 DATA 169, 193     : REM $A9, $C1          -LDA #ASC "A"
150 DATA 32, 237, 253 : REM $20, $ED, $FD     -JSR 65005
160 DATA 96           : REM $60               - RTS
170 HOME
180 CALL CODE
```

You may well be wondering just what new value should be used with the HIMEM: statement. Well, this will depend on the length of your machine language program. The formula is simply:

38400 - (256 * (INT (PROGRAM LENGTH/256) + 1))

(where 38400 is the default value of HIMEM). In the above example I decided to reserve 2 pages of memory (512 bytes).

Finally, there is one more area in the Apple's memory where short machine language programs can be stored. This is the input buffer. The input buffer consists of page 2 of memory and can accommodate 256 bytes. Generally, any information that is typed on the Apple's keyboard is first placed in the input buffer. Since most entries are short, it is usually safe to use at least half of the available space in the input buffer for program storage. For programs that are longer than 208 bytes and shorter than about 400 bytes, a combination of the input buffer and page 3 are used, with routines and/or data that are only used once being placed in the input buffer. The one-time, or disposable part of the program is placed in the input buffer so no damage will be done to the rest of the program if it is accidentally overwritten by a long line of text typed in from the keyboard (see Figure 6.3). Program 3 uses this area.

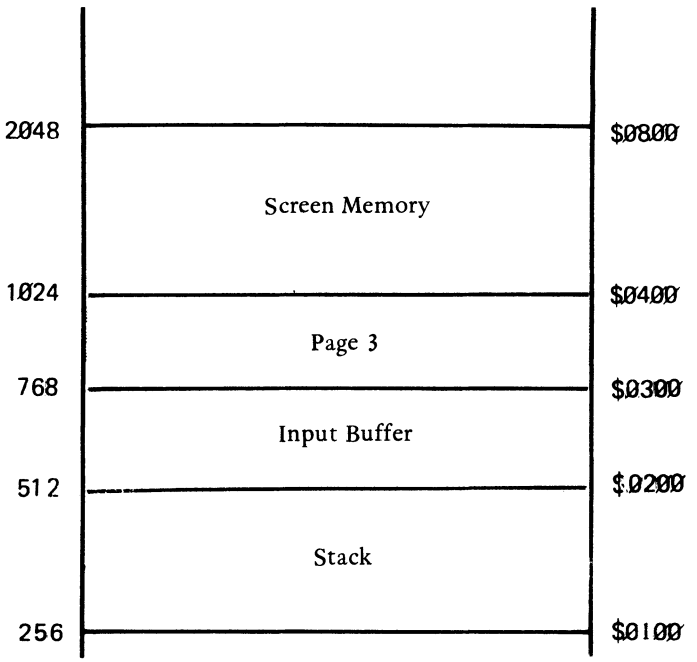


Figure 6.3 Place machine code in the input buffer.

Program 3

```
 10 REM ** PLACE M/C IN THE INPUT BUFFER **
 20 REM ** FROM $280 TO $2FF (640 TO 767) **
 30 CODE = 640
 40 FOR LOOP = 0 TO 5
 50       READ BYTE
 60       POKE CODE + LOOP,BYTE
 70 NEXT LOOP
 80 :
 90 REM ** M/C DATA **
100 DATA 169, 193     : REM $A9, $C1        -LDA #$C1
110 DATA 32, 237, 253 : REM $20, $ED, $FD   -JSR 65005
120 DATA 96           : REM $60             -RTS
130 HOME
140 CALL CODE
```

To summarize then, the following areas of memory can be considered 'safe' for machine code:

1. Page 3 ($300 to $3CF)
2. Above reset HIMEM.
3. In part of the input buffer if long lines aren't typed in from the keyboard.

Most Programs in this book use page 3 of memory.

ENTERING MACHINE CODE

The most obvious way of entering machine code is to write a program that just contains line after line of POKEs. Program 4 shows how this method can be used to produce a machine code program that switches on the inverse character mode which causes all characters printed to the screen by the computer to be shown as black on white.

Program 4

```
 10 REM ** INVERSE ON USING POKES **
 20 REM ** PLACE M/C IN PAGE 3 **
 30 POKE 768,169   : REM $A9          -LDA #$3F
 40 POKE 769,63    : REM $3F
 50 POKE 770,32    : REM $20          -JSR 62073
 60 POKE 771,121   : REM $79
 70 POKE 772,242   : REM $F2
 80 POKE 773,96    : REM $60          -RTS
 90 HOME
100 CALL 768
```

To see the effect of this, add the following lines from Program 1 to write an 'A' on the screen:

```
 72 POKE 773,169   : REM $A9          -LDA #$C1
 74 POKE 774,193   : REM $C1
 76 POKE 775,32    : REM $20          -JSR 65005
 78 POKE 776,237   : REM $ED
 80 POKE 777,253   : REM $FD
 82 POKE 778,96    : REM $60          -RTS
```

To get out of the inverse mode all you have to do is type POKE 769,255 and then CALL 768 or simply type NORMAL from Applesoft BASIC.

As you may be beginning to appreciate, entering machine code in this manner is somewhat laborious, particularly when it is a very long program. In the earlier programs the machine code was placed in a series of DATA statements, which were subsequently READ from within a FOR...NEXT loop and then POKEd into memory using the loop counter (LOOP) as an offset from the base address defined by CODE. This ensures that each byte is placed into consecutive memory locations.

Notice also, that in each program, every machine code operation was placed in a separate DATA statement, and was accompanied by a REM statement giving the same information in both the hex and mnemonic formats, for example:

```
100 DATA 169, 193    :  REM $A9, $C1        — LDA #ASC ("A")
```

The REM items are included for flexibility. Each of the programs can be entered and RUN exactly as it stands, thus allowing you to get programming in machine code straightaway; however, if at some time in the future you invest in an assembler program then you'll need to know the mnemonic versions. (The hex values are included for a reason that will soon become apparent!)

It is a good idea to get into the habit of including this type of REM statement into your own programs simply because it adds to the program's readability. Imagine being presented with a program that includes a single DATA statement:

```
100 DATA 169,14,32,208,241,169,146,32,202,241,96
```

It's not particularly clear what's being performed, and if you need to debug it, well...!

One final point regarding the loop count. This should be set to the total number of data bytes minus one. Remember that the loop counter itself must always start at '0' to ensure that the very first byte is placed at the address specified by CODE-CODE+LOOP = 768+0 = 768 (if CODE = 768)

THE HEX LOADER PROGRAM

An easier method of entering machine code is to use a monitor or hex loader program. This is a program which allows the machine code to be entered as a series of hex numbers. Program 5 is a simple example.

Program 5

```
 10 REM ** APPLE HEX LOADER **
 20 HOME
 30 HTAB (12)
 40 PRINT"APPLE HEX LOADER"
 50 PRINT:PRINT
 60 INPUT"ASSEMBLY ADDRESS: ";A$
 70 ADDR = VAL (A$)
 75 PRINT
 80 REM ** MAIN PROGRAM LOOP **
 90 PRINT ADDR;": $";
100 REM ** GET HIGH NIBBLE OF BYTE **
110 GOSUB 2000
120 HIGH = NUM
130 PRINT Z$;
```

```
140 REM ** GET LOW NIBBLE OF BYTE **
150 GOSUB 2000
160 LOW = NUM
170 PRINT Z$
180 REM ** CALCULATE BYTE AND UPDATE **
190 BYTE = HIGH * 16 + LOW
200 POKE ADDR,BYTE
210 ADDR = ADDR + 1
220 GOTO 80
500 REM ** DATA ENTRY AND CONVERSION **
2000 GET Z$
2010 IF Z$ = "S" THEN PRINT"STOP":END
2020 IF Z$ > "F" THEN 2000
2030 IF Z$ = "A" THEN NUM = 10:RETURN
2040 IF Z$ = "B" THEN NUM = 11:RETURN
2050 IF Z$ = "C" THEN NUM = 12:RETURN
2060 IF Z$ = "D" THEN NUM = 13:RETURN
2070 IF Z$ = "E" THEN NUM = 14:RETURN
2080 IF Z$ = "F" THEN NUM = 15:RETURN
2090 IF Z$ < "0" OR Z$ > "9" AND Z$ < "A" THEN 2000
2100 NUM = VAL (Z$):RETURN
```

The functions performed by each line in the program are as follows:

Line 20 Clears the screen.
Line 30 Spaces over 12 positions for next PRINT statement.
Line 40 Prints out heading.
Line 50 Prints two line feeds to skip two lines.
Line 60 Gets the starting address of the machine code.
Line 70 Converts the inputted string to a numeric value.
Line 75 Prints a line feed.
Line 90 Prints the address and ": $".
Line 110 Gets the high nibble of the hex byte.
Line 120 Saves its value in HIGH.
Line 130 Prints the high nibble.
Line 150 Gets the low nibble of the hex byte.
Line 160 Saves its value in LOW.
Line 170 Prints the low nibble.
Line 180 Calculates the value of the byte.
Line 200 POKEs the value of the byte into memory.
Line 210 Increments the memory counter.
Line 220 Repeats the entire process.
Line 2000 Wait for a key to be pressed.
Line 2010 If it's an S print the word STOP and end the program.
Line 2020 If it's greater than F ignore it and wait for another key to be pressed.
Line 2030-2080 If it's in the range of A-F declare its value and return.
Line 2090 Checks to see if the key pressed was a number or letter in the desired range. If it's not, it's ignored and the program waits for another key to be pressed.
Line 2100 A number was pressed convert string to a numeric value and return.

Enter and RUN the program. After it displays the heading you are asked to input an Assembly Address. This is simply the address of the location in memory where you want to start storing your machine language code. This is normally the value you would assign to CODE and it should be entered as a decimal value. On pressing RETURN, the first program address is displayed followed by a dollar sign ($). All you now have to do is type in the hex digits. After you type the second digit, the value of the byte is calculated and then POKEd into memory. The next address is then

displayed. The program checks for (and ignores) non-hex characters. To quit the program at any time, just type 'S' (for STOP). Figure 6.4 shows the result of a typical run of the program. Once the machine code has been entered, it can be tested by using the CALL command followed by the address (in decimal) of the first byte of the machine code program. In this case, the program can be run by typing 'CALL 768'.

```
                    APPLE HEX LOADER


ASSEMBLY ADDRESS: 768


                        768: $A9
                        769: $C1
                        770: $20
                        771: $ED
                        773: $60
                        774: $STOP
```

Fig. 6.4 A sample run of the Apple Hex Loader program.

CALLING MACHINE CODE

To execute a machine code program, the Applesoft BASIC statement 'CALL' is used. To tell the Applesoft BASIC interpreter just where the machine code is located, the CALL statement must be followed by a label or address. So to execute machine code generated by the assembly language program type in either:

```
CALL CODE
```

which is the label name that marks the start of the assembly language program, or:

```
CALL 768
```

which is the starting address of the machine code itself.

SAVING IT OUT TO DISK

It is a very good idea, as a matter of routine, to get into the habit of saving your machine code programs to disk before you actually run them. This may seem a little strange because you normally would not do this in BASIC until you had RUN, tested and debugged the program. The trouble with running a machine code program for the first time, however, is that if it does contain any bugs, it will almost certainly cause

the Apple // to 'hang up', and frequently the only way out of this is to switch the computer off and then back on, and start all over again. If your machine code does fail in this way, and you've saved it out onto the disk, all you have to do is reload it in and swat the bug out!

Once the program is fully debugged it is possible to save just the machine code if so required. We shall look at how to do this in Chapter 20.

THE APPLE ROMs

Supplied pre-packaged with every Apple // computer is a set of machine code routines which are available for use from within machine language programs. These routines belong to part of Apple's operating system which is located in the Apple's memory between locations $D000 and $FFFF. There are many useful routines in the Apple ROMs, but for the present we need only concern ourselves with the more commonly used ones which are summarized in Table 6.1.

Table 6.1

Routine	Address	Operation
CHRGET	177 ($00B1)	Get a character & increment ptr.
CHRGOT	183 ($00B7)	Get a character, no increment
RUN	54630 ($D566)	Run BASIC program
NEWSTT	55250 ($D7D2)	Execute new BASIC statement
LINGET	55820 ($DA0C)	Read a line number
CRDO	56059 ($DAFB)	Print carriage return
STROUT	56122 ($DB3A)	Print string pointed to by Y,A
OUTSPC	56151 ($DB57)	Print space
OUTDO	56156 ($DB5C)	Print character in A
FRMNUM	56679 ($DD67)	Evaluate formula, numbers only
FRMEVL	56699 ($DD7B)	Evaluate formula uses CHRGET
GETADR	59218 ($E752)	Convert number in FAC to integer
LINPRT	60708 ($ED24)	Print 2 bytes in X,A as integer
MINASM	63078 ($F666)	Enter miniassembler if present
PLOT	63488 ($F800)	Plot a point
CLREOL	64668 ($FC9C)	Clear to end of line
RDKEY	64780 ($FD0C)	Wait for key press
GETLN	64874 ($FD6A)	Input a line
KEYIN	64795 ($FD1B)	Input a character
COUT	65005 ($FDED)	Output a character

We have already used the COUT routine several times to write the character in the accumulator to the screen. The instruction takes the form JSR 65005 (or JSR $FDED). The mnemonic 'JSR' simply tells the 6502 microprocessor to jump to the address given, and then come back to where it was when it is finished so that the next instruction can be executed. This is known as a 'subroutine' jump — which we shall look at in detail in Chapter 14.

7 Status Symbols

THE STATUS REGISTER

The Status register is unlike the various 'other' registers of the 6502. When using it we are not really concerned with the actual hex value it contains, but more with the condition or state of the individual bits. These individual bits are used to denote, or *flag*, certain conditions as and when they occur during the course of a program. Of the register's eight bits, only seven are used — the remaining bit (bit 5) is permanently set. (In other words it always contains a 1).

Figure 7.1 shows the position of the various flags, each of which is now described in detail.

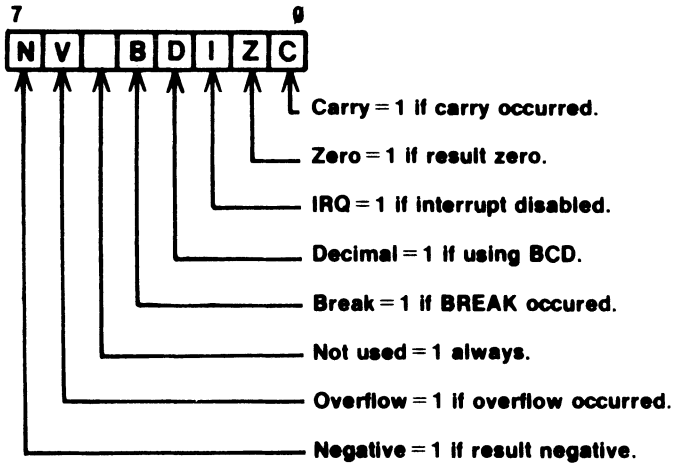


Figure 7.1 Status register flags.

Bit 7: The Negative flag (N)

In signed binary, the Negative flag is used to determine the sign of the number. If the flag is set (N=1) the result is negative. If the flag is clear (N=0) the result is positive.

However, a whole host of other instructions condition this particular flag , including all the arithmetic and logical functions. In general, the most significant bit of the result of an operation is copied directly into the N flag.

Consider the following two operations:

```
LDA #$80          \  load the accumulator with $80
```

This will set the Negative flag (N=1) because $80 = 10000000 in binary. Alternatively:

```
LDA #$7F          \  load the accumulator with $7F
```

will clear the Negative flag (N=0) because $7F = 01000000 in binary. There are two instructions which will act on the state of the N flag — these are:

```
BMI    Branch on minus (N=1)
BPL    Branch on plus (N=0)
```

More on these later.

Bit 6: The Overflow Flag (V)

This flag is probably the least used of all the status register flags. It is used to indicate if a carry occurred from bit 6 during an addition, or if a borrow occurred from bit 6 in a subtraction. If either of these events took place the flag is set (V=1).

Look at the following two examples:

First, $09 + $07:

```
($09)      0 0 0 0 1 0 0 1
($07)    + 0 0 0 0 0 1 1 1
           -----------------
($10)      0 0 0 1 0 0 0 0
           |
           +-----No overflow from bit 6 therefore V = 0.
```

Second, $7F + $01

```
($7F)      0 1 1 1 1 1 1 1
($01)    + 0 0 0 0 0 0 0 1
           -----------------
($80)      1 0 0 0 0 0 0 0
           |
           +-----Overflow has occurred from bit 6 therefore V = 1.
```

If we were using signed binary, the addition would give a result of -128, which is of course incorrect. However, this fact is flagged and so the result can be corrected as required.

Bit 5

This bit is not used and is permanently set.

Bit 4: The Break Flag (B)

This flag is set whenever a BREAK occurs, otherwise it will remain clear. This may seem a bit odd at first, because surely we will know when a BREAK occurs. However, it is possible to generate a BREAK externally by something called an *Interrupt*, and this flag is used to help distinguish between these 'BREAKs'.

Bit 3: The Decimal Flag (D)

This flag tells the processor just what type of arithmetic is being used. If it is cleared (by CLD), as is usual, then normal hexadecimal operation occurs. If set (by SED) all values will be interpreted as Binary Coded Decimal.

Bit 2: The Interrupt Flag (I)

We mentioned interrupts above in the description of the Break flag, and they will be looked at in more detail in Chapter 20. Suffice it to say now, that the flag is set (I=1) when the IRQ interrupt is disabled, and clear (I=0) when IRQ interrupts are permitted.

Bit 1: The Zero Flag (Z)

As its name implies, the flag is used to show whether or not the result of an operation is zero. If the result is zero the flag is set (Z=1), otherwise it is cleared (Z=0). It is true to say that the Zero flag is conditioned by the same instructions as the Negative flag. Executing:

```
LDA #0          \  load accumulator with zero
```

will set the Zero flag (Z=1) but:

```
LDX #$FA        \  load the X-register with $FA
```

will clear the Zero flag (Z=0).

Bit 0: The Carry Flag (C)

We have already seen that adding two bytes together can result in carries occurring from one bit to another. What happens if the carry is generated by the most significant bits of an addition?

For example, when adding $FF + $80:

```
($FF)      1 1 1 1 1 1 1 1
($80)    + 1 0 0 0 0 0 0 0
($7F)    (1)0 1 1 1 1 1 1 1
           ↑
           Carry over from bits 7
```

the result is just too large for eight bits, an extra ninth bit is required. The Carry flag acts as the ninth bit.

If the Carry flag is clear at the start of an addition (C=0) and set on the completion (C=1) the result is greater than 255. It follows that if the Carry flag is set (C=1) before a subtraction and clear on completion (C=0), the value being subtracted was larger than the original value. Two instructions are available for direct use on the Carry flag:

CLC	Clear Carry flag (C=0)
SEC	Set Carry flag (C=1)

Two instructions are provided to act on the condition of the Carry flag:

BCC	Branch on Carry clear (C=0)
BCS	Branch on Carry Set (C=1)

8 Addressing Modes I

The 6502 and 65C02 have quite small instruction sets when compared with some of their fellow microprocessors—in fact the 6502 has a basic clique of just 56 instructions, while the 65C02 has only 8 more instructions, for a total of 64. However, many of these can be used in a variety of ways, which effectively increases the range of operations to 151 for the 6502 and 178 for the 65C02. The way in which these operations are interpreted is determined by the *addressing mode* used. The following examples are in hex format.

Addressing mode	Mnemonic	Opcode	Operand(s)
Immediate	LDA #255	A9	FF
Zero page	LDA $FB	A5	FB
Zero page indexed	LDA $FB,X	B5	FB
Absolute	LDA $CD00	AD	00 CD
Indirect pre-indexed	LDA ($FB,X)	A1	FB
Indirect post-indexed	LDA ($FB),Y	B1	FB
Absolute indexed	LDA $CD00,X	BD	00 CD

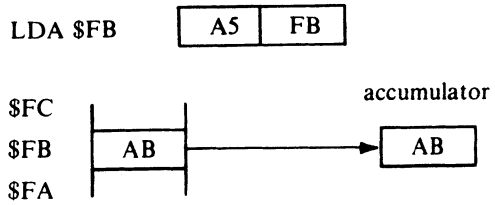
All seven of these instructions load the accumulator—but in each case the data loaded is obtained from a different source as defined by the opcode. This, as you may have noticed, is different in each case.

For the time being we shall look only at the first two of these addressing modes, immediate and zero page, both of which we have used several times already.

ZERO PAGE ADDRESSING

Zero page addressing is used to specify an address in the first 256 bytes of RAM where data which has been loaded into a specific register may be located. Because the high byte of the address is always $00 it is omitted, and therefore the instruction and address require just two bytes of memory.

Operation: DATA $A5, $FB



In the example, LDA $FB, the contents of location $FB (which in this case consists of the value $AB) are loaded into the accumulator.

The use of zero page needs some special care as this area is used by the Applesoft BASIC interpreter as a scratchpad for storing addresses and performing calculations. However, Apple's designers have kept a few bytes clear for us to use as we please. These include locations $6-$9, $18-$1F, $CE-$CF, $D6-$D7, $E3, $EB-$EF and $F9-$FF. The instructions associated with zero page addressing are shown in Table 8.1.

Table 8.1

Zero page addressing instructions for the 6502

ADC	Add with carry	LDY	Load Y register
AND	Logical AND	LSR	Logical shift right
ASL	Arithmetic shift left	ORA	Logical OR
BIT	Bit test	ROL	Rotate left
CMP	Compare accumulator	ROR	Rotate right
CPX	Compare X register	SBC	Subtract with carry
CPY	Compare Y register	STA	Store accumulator
DEC	Decrement memory	STX	Store X register
EOR	Logical EOR	STY	Store Y register
INC	Increment memory	STZ	Store zero in memory
LDA	Load accumulator	TRB	Test and reset bits
LDX	Load X register	TSB	Test and set bits

IMMEDIATE ADDRESSING

This form of addressing is used to load the accumulator or index registers with a specific value which is known at the time of writing the program. The 6502 knows from the opcode that the byte following it is in actual fact data and not an address. However, to remind us of the fact, and to assist us when we are writing the program, we can precede the data byte with a number sign '#' (this shares the '3' key on the Apple's keyboard). Only single byte values can be specified because the register size is limited to just eight bits.

If we wish our machine code program to load the accumulator with 255, we can include the following two-byte sequence in our program:

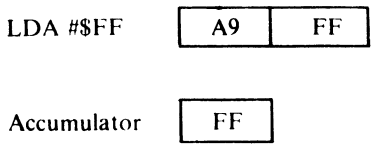
```
DATA 169,255        : REM $A9,$FF      —LDA #$FF
```

where 169 ($A9) is the 'load the accumulator immediate' code.Similarly, the X and Y registers can be loaded immediately with:

```
DATA 162,65         : REM $A2,$41      —LDX #ASC("A")
DATA 160,7          : REM $A0,$07      —LDY #$07
```

where 162 ($A2) and 160 ($A0) are the immediate codes for loading the X and Y registers, and 65 ($41) is the ASCII code for the letter A.

Operation:



Program 6 uses both zero page and immediate addressing to place an exclamation point on the screen.

Program 6

```
 10  REM ** ZERO PAGE AND IMMEDIATE ADDRESSING **
 20  CODE = 768
 30  FOR LOOP = 0 TO 9
 40      READ BYTE
 50      POKE CODE + LOOP, BYTE
 60  NEXT LOOP
 70  :
 80  REM ** M/C DATA **
 90  DATA 162,161        : REM $A2,$A1      -LDX #ASC"!"
100  DATA 134,251        : REM $86,$FB      -STX $FB
110  DATA 165,251        : REM $A5,$FB      -LDA $FB
120  DATA 32,237,253     : REM $20,$ED,$FD  -JSR $FDED
130  DATA 96             : REM $60          -RTS
140  :
150  CALL CODE
```

The meaning of each line is as follows:

Line 20	Assemble in 'free' RAM at $300.
Line 30-60	READ and POKE machine code.
Line 90	Load X register with ASCII code for '!'.
Line 100	Store X register contents in location $FB.
Line 110	Load accumulator with contents of $FB.
Line 120	Jump to subroutine to print accumulator's contents.
Line 130	Return to Applesoft.
Line 150	Call machine code.

9 Bits and Bytes

LOAD, STORE AND TRANSFER

To enable memory and register contents to be altered and manipulated, three sets of instructions are provided.

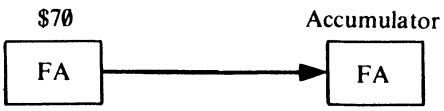
Load instructions

The process of placing memory contents into a register is known as *loading*, some examples of which we have already seen. To recap however, these are the three load instructions:

LDA	Load accumulator
LDX	Load X register
LDY	Load Y register

All of these instructions may be used with immediate addressing, but when dealing with memory locations, it is more correct to say that the contents of the specified address are *copied* into the particular register, as the source location is not altered in any way.

For example, with LDA $70, the contents of location $70 (in this case FA) are copied into the accumulator and location $70 is not altered:



The Negative and Zero flags of the Status register are conditioned by the load operation.

Store instructions

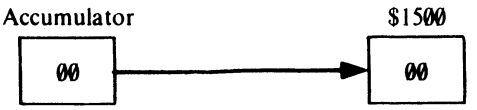
The reverse process of placing a register's contents into a memory location, is known as *storing*. There are three store instructions:

STA Store accumulator
STX Store X register
STY Store Y register
STZ Store a zero (65CO2 only)

The register value is unaltered and no flags are conditioned.

Example:

```
LDA #0
STA $1500
```



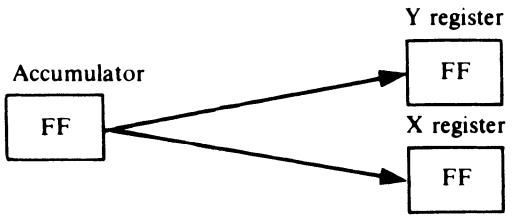
Transfer instructions

Instructions are provided to allow the contents of one register to be copied into another—this is known as *transferring*. The Negative and Zero flags are conditioned according to the data being transferred. There are four instructions controlling transfers between the index registers and the accumulator.

TXA Transfer X register to accumulator
TAX Transfer accumulator to X register
TYA Transfer Y register to accumulator
TAY Transfer accumulator to Y register

Example:

```
LDA #$FF
TAY
TAX
```



Unfortunately you cannot transfer directly between the X and Y registers, you have to use the accumulator as an intermediate store.

```
YTOX  TYA          \  Y into accumulator
      TAX          \  accumulator into X
```

Similarly:

```
XTOY  TXA          \  X into accumulator
      TAY          \  accumulator into Y
```

This form of single byte operation is known as *implied addressing* because the information is contained within the instruction itself.

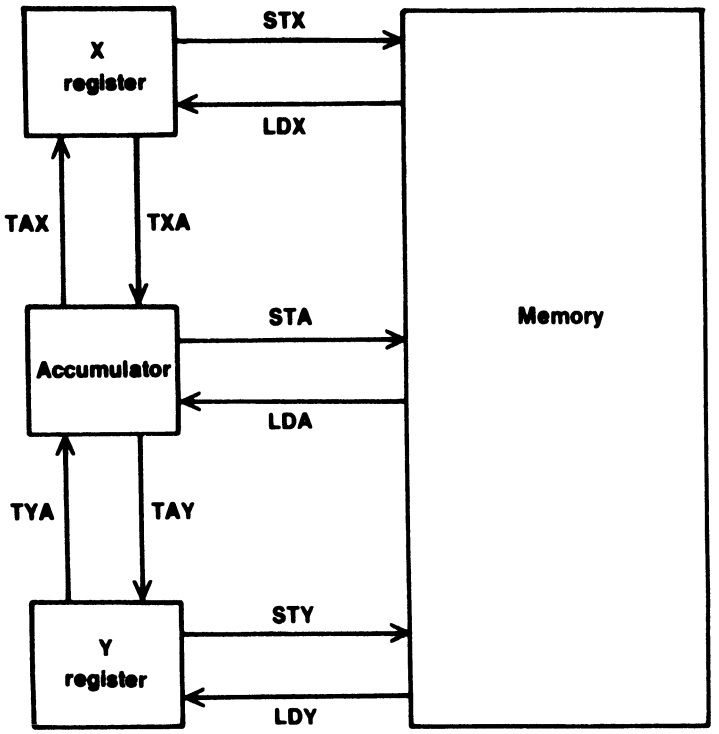


Figure 9.1 Load, store and transfer instruction flow.

PAGING MEMORY

We have seen that the Program Counter consists of two eight bit registers, giving a total of 16 bits. If all these bits are set, 11111111 11111111, the value obtained is 65535 or $FFFF. Therefore the maximum addressing range of the 6502 is $0000 through to $FFFF. This range of addresses is implemented as a series of pages and the page number is given by the contents of PCH. It follows that PCL holds the address of the location on that particular page.

As Figure 9.2 illustrates, each page of memory can be likened to a page of a book. This book, called "Apple's Memory", has 256 pages labelled in hex format from $00 to $FF. Each individual page is ruled into 256 lines which in turn are labelled (from top to bottom) $00 to $FF.

Thus the address $FFFF refers to line $FF on page $FF, the very last location in the Apple's memory map! Unlike conventional books, "Apple's Memory" begins with page $00 which is known more affectionately as *zero page*. Owing to the 6502's design zero page is very important, as we shall see when we take a further look at addressing modes in the next chapter.

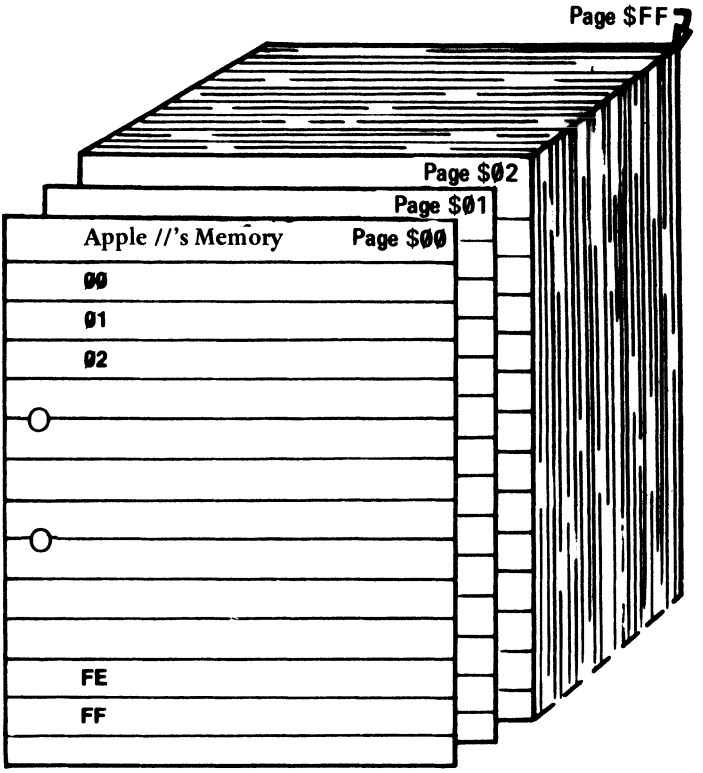


Figure 9.2 Pages of 'Apple //'s Memory'

Although we have referred to the Apple's memory map as a series of pages, it is more frequently talked of in terms of 'K'. The term 'K' is short for kilo, but unlike its metric counterpart, one kilo of memory, or a kilobyte, consists of 1024 bytes and not 1000 bytes. This slightly higher value is chosen because it is divisible by 256 and corresponds to exactly four pages of memory (4 x 256 = 1024). The total memory map therefore encompasses 64K because 65536/1024 = 64!

10 Arithmetic in Assembler

We can now put some of the basic principles we have encountered in the opening chapters to some more serious use—the addition and subtraction of numbers. These two procedures are fundamental to assembly language and will generally find their way into most programs.

ADDITION

Two instructions facilitate addition, they are:

CLC	Clear Carry flag
ADC	Add with carry

The first of these instructions, CLC, simply clears the Carry flag (C = 0). This will generally be performed at the very onset of addition, because the actual addition instruction, ADC, produces the sum of the accumulator, the memory byte referenced and the Carry flag. The reason for doing this will become clearer after we have looked at some simple addition programs. Enter Program 7.

Program 7

```
 10  REM ** SIMPLE ADD **
 20  CODE = 768
 30  FOR LOOP =0 TO 7
 40      READ BYTE
 50      POKE CODE + LOOP, BYTE
 60  NEXT LOOP
 70  :
 80  REM ** M/C DATA **
 90  DATA 24           : REM $18          -CLC
100  DATA 169,7        : REM $A9, $07     -LDA #$07
110  DATA 105,3        : REM $69, $03     -ADC #$03
120  DATA 133,251      : REM $85, $FB     -STA $FB
130  DATA 96           : REM $60          -RTS
140  :
```

```
150 CALL CODE
160 PRINT "ANSWER IS :";
170 PRINT PEEK (251)
```

As you can see, this program loads 7 into the accumulator using immediate addressing. Immediate addressing is used again in line 110 to add 3 to the accumulator value. The result, which is in the accumulator, is then stored at location 251. Line 150 executes the assembled machine code, and the result (if you're quick with your fingers you'll know its 10!) is printed out. RUN the program to see its effect then try substituting your own values in lines 100 and 110.

Re-type line 90 thus:

```
90 DATA 56          : REM $38          -SEC
```

As you probably realize, the Carry flag will now be set (C = 1) when the program is next executed. Reset lines 100 and 110 (if you have altered them), and RUN the program again. The result is now 11. The reason being that the Carry flag's value is taken into consideration during ADC (add with carry) and this time its value is 1.

	Accumulator	+	memory	+	carry	=	result
CLC	7	+	3	+	0	=	10
SEC	7	+	3	+	1	=	11

Again you might like to try your own immediate values—you'll find the result is always one greater than expected.

This program is quite wasteful both in terms of memory used and time taken for execution. If we know the values to be added together beforehand, then it is more efficient to add them together first. The machine code part of the program can then be incorporated into just two lines:

```
LDA #10          \  place 10 (7 + 3) into accumulator
STA $FB          \  store accumulator
```

Program 8 is a general purpose single byte addition program.

Program 8

```
 10 REM ** SINGLE BYTE ADD **
 20 CODE = 768
 30 FOR LOOP = 0 TO 7
 40     READ BYTE
 50     POKE CODE + LOOP, BYTE
 60 NEXT LOOP
 70 :
 80 REM ** M/C DATA **
 90 DATA 24          : REM $18          -CLC
100 DATA 165,251     : REM $A5, $FB     -LDA $FB
110 DATA 101,252     : REM $65, $FC     -ADC $FC
```

```
120 DATA 133,253        : REM $85, $FD      -STA $FD
130 DATA 96             : REM $60           -RTS
140 :
150 HOME
160 PRINT "SINGLE BYTE ADD DEMO"
170 PRINT : PRINT
180 INPUT "FIRST NUMBER";A
190 INPUT "SECOND NUMBER";B
200 POKE 251, A : POKE 252, B
210 CALL CODE
220 PRINT "ANSWER IS :";
230 PRINT PEEK (253)
```

RUN the program a few times entering low numerical values in response to the program's prompts.

Now enter 128 and 128 as your inputs. The result is 0. Why? The reason is that the answer, 256, is too big to be held in a single byte:

```
  128          $80          10000000
+ 128        + $80        + 10000000
  ---          ---          --------
  256          $100        (1)00000000
```

and as can be seen, a carry has been produced by the bit overflow from adding the two most significant bits. As the Carry flag was initially cleared before the addition, it will now be set, signaling the fact that the result is too large for a single byte.

This principle is used when summing multibyte numbers, and is illustrated by Program 9, which adds two double byte numbers.

Program 9

```
 10 REM ** DOUBLE BYTE ADD **
 20 CODE = 768
 30 FOR LOOP = 0 TO 13
 40     READ BYTE
 50     POKE CODE + LOOP, BYTE
 60 NEXT LOOP
 70 :
 80 REM ** M/C DATA **
 90 DATA 24              : REM $18           -CLC
100 DATA 165,251         : REM $A5, $FB      -LDA $FB
110 DATA 101,253         : REM $65, $FD      -ADC $FD
120 DATA 133,251         : REM $85, $FB      -STA $FB
130 DATA 165,252         : REM $A5, $FC      -LDA $FC
140 DATA 101,254         : REM $65, $FE      -ADC $FE
150 DATA 133,252         : REM $85, $FC      -STA $FC
160 DATA 96              : REM $60           -RTS
170 :
180 HOME
190 PRINT "DOUBLE BYTE ADD DEMO"
200 PRINT : PRINT
210 INPUT "FIRST NUMBER";A
220 REM CALCULATE HIGH AND LOW BYTE
230 AH = 1NT(A/256)
240 AL = A - (AH * 256)
```

```
250 INPUT "SECOND NUMBER";B
260 REM CALCULATE HIGH AND LOW BYTE
270 BH = INT(B/256)
280 BL = B - (BH * 256)
290 POKE 251, AL : POKE 252, AH
300 POKE 253, BL : POKE 254, BH
310 CALL CODE
320 LOW = PEEK(251) : HIGH = PEEK (252)
330 RESULT = HIGH * 256 + LOW
340 PRINT "ANSWER IS :";
350 PRINT RESULT
```

The meaning of each line is as follows:

Line 20	Assemble code in 'free' RAM from $300.
Lines 30-60	READ and POKE machine code data.
Line 90	Clear Carry flag.
Line 100	Get low byte of first number, AL.
Line 110	Add it to low byte of second number, BL.
Line 120	Store low byte of result.
Line 130	Get high byte of first number, AH.
Line 140	Add it to high byte of second number, BH.
Line 150	Store high byte of result.
Line 160	Return to BASIC.
Lines 180-190	Clear screen and print title.
Line 210	Input first number.
Line 230	Calculate high byte value of A.
Line 240	Calculate low byte value of A.
Line 250	Input second number.
Line 270	Calculate high byte value of B.
Line 280	Calculate low byte value of B.
Lines 290-300	POKE high and low byte values of A, B into memory.
Line 310	Execute machine code.
Line 320	Get low and high bytes of the result.
Line 330	Calculate result.
Lines 340-350	Print result.

This routine will produce correct results for any two numbers whose sum is not greater than 65535 ($FFFF) which is the highest numerical value that can be held in two bytes of memory.

Note that the Carry flag is cleared at the onset of the machine code itself. If any carry should occur when adding the two low bytes together, it will be transferred over to the addition of the two high bytes.

SUBTRACTION

The two associated instructions are:

SEC	Set Carry flag
SBC	Subtract, borrowing carry

The operation of subtracting one number from another (or finding their difference) is the reverse of that used in the preceding addition examples. Firstly, the Carry flag is set (C = 1) with SEC, and then the specified value is subtracted from the accumulator using SBC. The result of the subtraction is returned in the accumulator. The following program performs a single byte subtraction:

Program 10

```
 10 REM ** SIMPLE SUBTRACTION **
 20 CODE = 768
 30 FOR LOOP =0 TO 7
 40     READ BYTE
 50     POKE CODE + LOOP, BYTE
 60 NEXT LOOP
 70 :
 80 REM ** M/C DATA **
 90 DATA 56          : REM $38          -SEC
100 DATA 165,251     : REM $A5, $FB     -LDA $FB
110 DATA 229,252     : REM $E5, $FC     -SBC $FC
120 DATA 133,253     : REM $85, $FD     -STA $FD
130 DATA 96          : REM $60          -RTS
140 :
150 HOME
160 INPUT "HIGHEST NUMBER";A
170 INPUT "LOWEST NUMBER";B
180 POKE 251, A : POKE 252, B
190 CALL CODE
200 PRINT "ANSWER IS ";
210 PRINT PEEK(253)
```

The meaning of each line is as follows:

Lines 20-60	Assemble machine code.
Line 90	Set Carry flag.
Line 100	Load high number into the accumulator.
Line 110	Subtract contents of $FC from it.
Line 120	Save result in $FD.
Line 130	Back to BASIC.
Lines 160-170	Get two values.
Line 180	POKE them into zero page.
Line 190	Call machine code.
Lines 200-210	Print the answer.

RUN the program and input your own values to see the results.

You may well be wondering why the Carry flag is set before a subtraction rather than cleared. Referring back to Chapter 3, you will recall that the subtraction there was performed by adding the two's complement value. This is found by first inverting all the bits to obtain the one's complement, and then adding 1. The 6502 obtained the 1 to be added to the one's complement form, from the Carry flag. Thus we can say:

1. If the Carry flag is set after SBC, the result is positive or zero.
2. If the Carry flag is clear after SBC, the result is negative and a borrow has occurred.

Try changing line 90 to DATA 24: REM $18—CLC and re-RUN the program. Now your results are one less than expected — the reason being that the two's complement was never obtained by the 6502, because only a '0' was available in the Carry flag to be added to the one's complement value.

To subtract double byte numbers the Carry flag is set at the entry to the routine, and the relative bytes are subtracted and stored. The resulting program looks something like this:

Program 11

```
 10  REM ** DOUBLE BYTE SUBTRACT10N **
 20  CODE = 768
 30  FOR LOOP = 0 TO 13
 40      READ BYTE
 50      POKE CODE + LOOP, BYTE
 60  NEXT LOOP
 70  :
 80  REM ** M/C DATA **
 90  DATA 56          : REM $38          —SEC
100  DATA 165,251     : REM $A5, $FB     —LDA $FB
110  DATA 229.253     : REM $E5, $FD     —SBC $FD
120  DATA 133,251     : REM $85, $FB     —STA $FB
130  DATA 165,252     : REM $A5, $FC     —LDA $FC
140  DATA 229,254     : REM $E5, $FE     —SBC $FE
150  DATA 133,252     : REM $85, $FC     —STA $FC
160  DATA 96          : REM $60          —RTS
170  :
180  HOME
190  INPUT "HIGHEST NUMBER";A
200  INPUT "LOWEST NUMBER";B
210  REM CALCULATE HIGH AND LOW BYTES
220  AH = INT (A / 256)
230  AL = A - (AH * 256)
240  BH = INT (B / 256)
250  BL = B - (BH * 256)
260  POKE 251, AL : POKE 252, AH
270  POKE 253, BL : POKE 254, BH
280  CALL CODE
290  LOW = PEEK (251) : High = PEEK (252)
300  RESULT = HIGH * 256 + LOW
310  PRINT "ANSWER IS ";
320  PRINT RESULT
```

The meaning of each line is as follows:

Lines 20-60	Assemble machine code.
Line 90	Set the Carry flag.
Line 100	Load low byte of high number into accumulator.
Line 110	Subtract low byte of low number from it.
Line 120	Save low byte of result in $FB.
Line 130	Load high byte of high number into accumulator.
Line 140	Subtract high byte of low number from it.
Line 150	Save high byte of result in $FC.
Line 160	Back to BASIC.
Lines 180-200	Get two numbers.
Lines 220-270	Calculate and store high and low bytes.

Line 280	Call machine code.
Lines 290-300	Calculate final result.
Lines 310-320	Print the answer.

NEGATION

The SBC instruction can be used to convert a number into its two's complement form. This is done by subtracting the number to be converted, from zero. The following program asks for a decimal value (less than 255) and prints its two's complement value in hex:

Program 12

```
 10  REM ** TWO'S COMPLEMENT CONVERTER **
 20  CODE = 768
 30  FOR LOOP =0 TO 7
 40      READ BYTE
 50      POKE CODE + LOOP, BYTE
 60  NEXT LOOP
 70  :
 80  REM ** M/C DATA **
 90  DATA 56          : REM $38          -SEC
100  DATA 169,0       : REM $A9, $00     -LDA #0
110  DATA 229,251     : REM $E5, $FB     -SBC $FB
120  DATA 133,252     : REM $85, $FC     -STA $FC
130  DATA 96          : REM $60          -RTS
140  :
150  HOME
160  INPUT "NUMBER TO BE CONVERTED";A
170  IF A > 255 THEN PRINT "ERROR" : GOTO 160
180  POKE 251, A
190  CALL CODE
200  PRINT "THE TWO'S COMPLEMENT VALUE IS :":
210  PRINT PEEK(252)
```

The meaning of each line is as follows:

Lines 20-60	Assemble machine code.
Line 90	Set the Carry flag.
Line 100	Load accumulator with 0.
Line 110	Subtract the contents of $FB from it.
Line 120	Save result in $FC.
Line 130	Back to BASIC.
Lines 150-160	Get number.
Line 170	Make sure it's less than 256.
Line 180	POKE number into $FB.
Line 190	Execute machine code.
Lines 200-210	Print result.

USING BCD

We can now investigate the use of BCD in assembly language programs. You will remember from Chapter 3 that 6 must be added to (or subtracted from) the result of an addition (or subtraction) whenever a transition occurs from 9 to 10 or vice versa. This causes the six unused binary combinations, normally used to represent A to F in hex, to be jumped over to produce the correct decimal result. You will be pleased to know that the 6502 will actually take care of this correction for you when it knows you are using BCD. But how does it know when you are using BCD? Well, you must find the condition by setting the Decimal flag in the Status register with:

SED Set Decimal flag (D = 1)

The corresponding flag clearing instruction is:

CLD Clear Decimal flag (D = 0)

Program 13 uses immediate addressing to perform a BCD addition.

Program 13

```
 10  REM ** SIMPLE BCD ADDITION **
 20  CODE = 768
 30  FOR LOOP = 0 TO 9
 40      READ BYTE
 50      POKE CODE + LOOP, BYTE
 60  NEXT LOOP
 70  :
 80  REM ** M/C DATA **
 90  DATA 248        : REM $F8          - SED
100  DATA 24         : REM $18          - CLC
110  DATA 169,9      : REM $A9, $09     - LDA #$09
120  DATA 105,5      : REM $69, $05     - ADC #$05
130  DATA 133,251    : REM $85, $FB     - STA $FB
140  DATA 216        : REM $D8          - CLD
150  DATA 96         : REM $60          - RTS
160  :
170  CALL CODE
180  PRINT PEEK (251)
```

As can be seen, BCD addition is no different from normal hex addition except for the three extra instructions SED, CLC and CLD. RUN the program. Unfortunately, the result is 20 and not 14 as we would expect. The reason for this is that the Applesoft print routine interprets the byte stored at location 251 ($FB) as a hex one and not a BCD one. Don't believe me eh? Well, the hex for 20 is $14 (told you!). Writing this in binary form we have:

0001 0100

This, as you will now realize, is the BCD binary for 14 BCD.

Chapter 21 contains a program that will print decimal numbers in hex form—which is really what we're after. This program can be used to output correct BCD values.

Similarly, a BCD subtraction would take the form:

```
SED             \  set decimal mode
SEC             \  set Carry flag
LDA VALUE       \  get first value
SBC NUMBER      \  subtract a number
STA RESULT      \  save the result
CLD             \  clear decimal mode
```

One final important point to remember regarding the Carry flag—when using BCD, the Carry flag signals a result greater than 99 during addition.

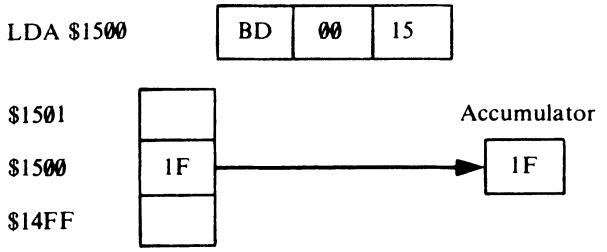
11 Addressing Modes II

Let us now take a second look at addressing modes. In the previous chapters we have seen how data can be obtained directly by an instruction using *immediate* addressing, or indirectly from a location in zero page using *zero page* addressing. We shall now see how two byte address locations can be accessed both directly and indirectly (through the all important zero page), and how whole blocks of memory can be manipulated using *indexed addressing.*

ABSOLUTE ADDRESSING

Absolute addressing works in exactly the same manner as zero page addressing, but it covers all memory locations outside zero page. The mnemonic is followed by two bytes which specify the address of the memory location (which can be anywhere in the range $100 to $FFFF).

Operation



As can be seen above, the operation code is followed by the address which, as always, is stored in reverse order low byte first. The contents of location $1500 are copied into the accumulator when the instruction is executed. Program 14 uses absolute addressing to place an inverse A on the screen; note that it is not printed but stored into screen memory.

Program 14

```
10  REM ** ABSOLUTE ADDRESSING **
20  CODE = 768
```

```
 30  FOR LOOP = 0 TO 5
 40      READ BYTE
 50      POKE CODE + LOOP, BYTE
 60  NEXT LOOP
 80  REM ** M/C DATA **
 90  DATA 169,1        :  REM $A9, $01       -LDA #$01
100  DATA 141,80,04    :  REM $8D, $50, $04  -STA 1104
110  DATA 96           :  REM $60            -RTS
120  :
130  HOME
140  PRINT : PRINT : PRINT
150  CALL CODE
```

The meaning of each line is as follows:

Lines 20-60	Assemble machine code.
Line 90	Load accumulator with display code for an inverse 'A'.
Line 100	Store A into screen memory.
Line 110	Back to BASIC.
Lines 130-140	Clear screen and move cursor down.
Line 150	Execute machine code.

The complete list of instructions associated with absolute addressing is shown in Table 11.1.

Table 11.1

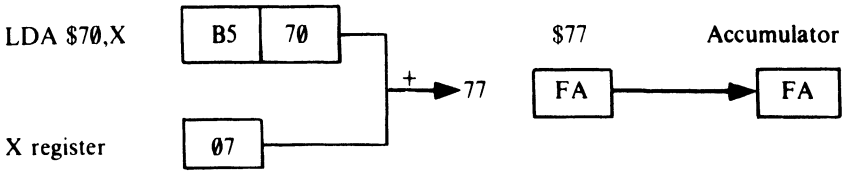
Absolute addressing instruction

ADC	Add with carry	LDX	Load X register
AND	Logical AND	LDY	Load Y register
ASL	Arithmetic shift left	LSR	Logical shift right
BIT	Bit test	ORA	Logical OR
CMP	Compare accumulator	ROL	Rotate left
CPX	Compare X register	ROR	Rotate right
CPY	Compare Y register	SBC	Subtract with carry
DEC	Decrement memory	STA	Store accumulator
EOR	Logical EOR	STX	Store X register
INC	Increment memory	STY	Store Y register
JMP	Jump	STZ	Store zero
JSR	Jump, save return	TRB	Test and reset
LDA	Load accumulator	TSB	Test and set

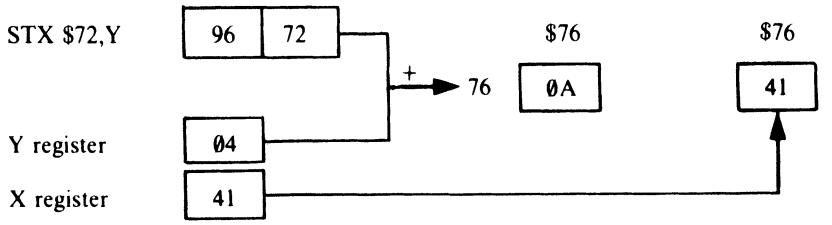
ZERO PAGE INDEXED ADDRESSING

In zero page indexed addressing, the actual address of the operand is calculated by adding the contents of either the X or Y register to the zero page address stated.

Operation:



The X register in this instance contains $07. This is added to the specified address, $70, to give the actual address, $77. The contents of location $77 (in this case FA) are then loaded into the accumulator. Similarly:



Here the Y register is used as an index to allow the contents of the X register to be stored in memory location $76. This address was obtained by adding the Y register's value, $04, to the specified value, $72. The original contents of location $76 (0A) are overwritten.

Note: The Y register can *only* be used to operate on the X register with instructions such as LDX $FB, Y. The instructions associated with zero page indexing are listed in Table 11.2.

Table 11.2

Zero page indexed addressing instructions

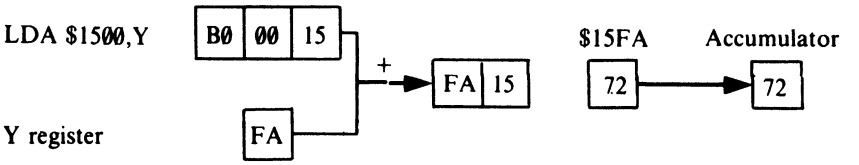
ADC	Add with carry	LDY	Load Y register
AND	Logical AND	LSR	Logical shift right
ASL	Arithmetic shift left	ORA	Logical OR
CMP	Compare	ROL	Rotate left
DEC	Decrement memory	ROR	Rotate right
EOR	Logical EOR	SBC	Subtract with carry
INC	Increment memory	STA	Store accumulator
LDA	Load accumulator	*STX	Store X register
*LDX	Load X register	STY	Store Y register
		STZ	Store zero

The * indicates the only commands which can use the Y register as an index. All other commands are for X register only.

ABSOLUTE INDEXED ADDRESSING

Absolute indexed addressing is like zero page indexed addressing except that the locations accessed are outside zero page. The X and Y registers may be used as required to operate with the accumulator, or each other.

Operation:



The Y register's contents ($FA) are added to the two byte address ($1500) to give effective address ($15FA). The following program demonstrates how absolute indexed addressing can be used to move a section of screen memory from one location to another.

Program 15

```
 10  REM ** ABSOLUTE INDEXED ADDRESSING **
 20  CODE = 768
 30  FOR LOOP = 0 TO 11
 40      READ BYTE
 50      POKE CODE + LOOP, BYTE
 60  NEXT LOOP
 70  :
 80  ** M/C DATA **
 90  DATA 162,32      : REM $A2, $20       — LDX #$20
100  DATA 189,0,4     : REM $BD, $00, $04  — LDA 1024, X
110  DATA 157,0,7     : REM $9D, $00, $07  — STA 1792, X
120  DATA 202         : REM $CA            — DEX
130  DATA 208,247     : REM $D0, $F7       — BNE -9
140  DATA 96          : REM $60            — RTS
150  :
160  HOME
170  PRINT " ABSOLUTE INDEXED ADDRESSING"
180  GET A$
190  IF A$ = "" THEN GOTO 180
200  CALL CODE
```

The meaning of each line is as follows:

Lines 20-60	Assemble machine code.
Line 90	Set X register count.
Line 100	Load accumulator with contents of location 1024 + X.
Line 110	Store accumulator's contents at 1792 + X.
Line 120	Decrement X register.
Line 130	IF X<>0 then go back.
Line 140	Back to BASIC.
Lines 160-170	Clear screen and print title.
Lines 180-190	Wait for a key to be pressed.
Line 200	Execute machine code.

When RUN, the message of line 170 is printed on to the screen. The program then waits for a key to be pressed before calling the machine code. The X register acts as the offset counter and is initialized in line 90. The byte at location 1024 + X is

loaded into the accumulator, and then stored back into screen memory at 1792 + X; in both instances absolute indexed addressing is used. Two new instructions are introduced in lines 120 and 130 and these will be examined in the next couple of chapters. Briefly though, DEX decreases the contents of the X register by one, and BNE tests to see if the X register has reached zero. If X is not zero, the specified jump takes place, causing the load/store procedure to be repeated with the new value of X. The instructions associated with absolute indexed addressing are shown in Table 11.3.

Table 11.3

Absolute indexed addressing instructions

*ADC	Add with carry	**LDX	Load X register
*AND	Logical AND	LDY	Load Y register
ASL	Arithmetic shift left	LSR	Logical shift right
*CMP	Compare memory	*ORA	Logical OR
DEC	Decrement memory	ROL	Rotate left
*EOR	Logical exclusive OR	ROR	Rotate right
INC	Increment memory	*SBC	Subtract with carry
*LDA	Load accumulator	*STA	Store accumulator
		STZ	Store zero

Unmarked commands are available with X register as index only. Commands marked * may use either register, whereas the one marked ** can only use the Y register.

INDIRECT ADDRESSING

Indirect addressing allows us to read or write to a memory address which is not known at the time of writing the program! Crazy? Not really, the program itself may calculate the actual address to be handled. Alternatively, a program may contain within it several tables of data which are all to be manipulated in a similar manner. Rather than writing a separate routine for each, a general purpose one can be developed, with the address operand being 'seeded' on each occasion the routine is called.

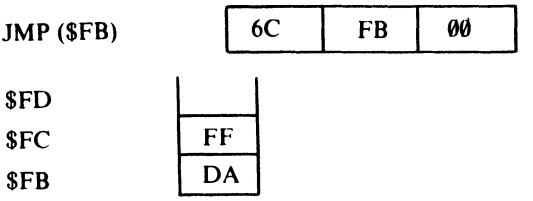
The beauty of indirect addressing is that it enables the whole of the Apple's memory map to be accessed with a single two byte instruction. To distinguish indirect addressing from other addressing modes, the operands must be enclosed in brackets.

Pure indexed addressing in only available to one instruction — the jump instruction — which is mnemonically represented by JMP. We will look at JMP's function in more detail during the course of Chapter 14, but suffice to say for now that it is the 6502's equivalent of BASIC's GOTO statement. (Though it does of course jump to an address rather than a line number.) A typical indirect jump instruction takes the form:

```
DATA 108, 251, 00 : REM $6C, $FB, $00 — JMP ($FB)
```

The address specified in the instruction is not the address jumped to, but is the address of the location where the jump address is stored. In other words, don't jump here but to the address stored here!

Operation:



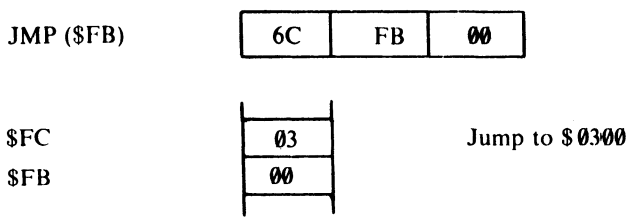
From the operational example we can see that location $FB contains the low byte of the address, and location $FC the high byte. These two locations, which act as temporary stores for the address, are known as a vector. Executing JMP ($FB) in this instance will cause the program to jump to the location $FFDA.

Program 16 illustrates the use of an indirect JMP to fill the screen with stars.

Program 16

```
 10  REM **INDIRECT JUMPING **
 20  CODE = 768
 30  FOR LOOP = 0 TO 15
 40      READ BYTE
 50      POKE CODE + LOOP, BYTE
 60  NEXT LOOP
 70  :
 80  REM ** M/C DATA **
 90  DATA 169,0       : REM $A9, $00       -LDA #$00
100  DATA 133,251     : REM $85, $FB       -STA $FB
110  DATA 169,3       : REM $A9, $03       -LDA #$03
120  DATA 133,252     : REM $85, $FC       -STA $FC
130  DATA 169,170     : REM $A9, $AA       -LDA #ASC"*"
140  DATA 32,237,253  : REM $20, $ED, $FD  -JSR $FDED
150  DATA 108,251,0   : REM $6C, $FB, $00  -JMP ($FB)
160  :
170  CALL CODE
```

Lines 90 to 120 set up a vector in zero page. Two of the free user bytes are loaded with the assembly address of the machine code, $0300 in this case. Line 130 places the ASCII code for the asterisk into the accumulator, and this is printed out using the monitor routine at $FDED (line 140). Finally the routine jumps back to the start via the zero page vector (line 150).



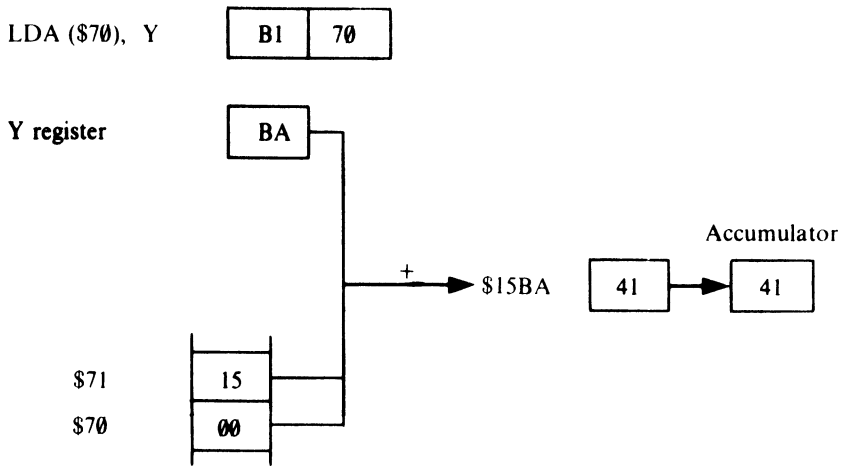
The program is now in a continuous loop and will carry on printing stars ad infinitum. I'm afraid pressing Control-C, will have no effect—that's for Applesoft BASIC only; you'll either have to re-boot the computer or turn it off and then back on again to return to any semblance of normality!

The Apple // itself uses indirect addressing extensively, particularly for directing control of input and output operations.

POST-INDEXED INDIRECT ADDRESSING

Post-indexed addressing is a little like absolute indexed addressing, but in this case, the base address is stored in a zero page vector which is accessed indirectly.

Operation:



In the example above, the base address is stored in the vector at $70 and $71. The contents of the Y register ($BA) are added to the address in the vector ($1500) to give the actual address ($15BA) of the data. It should be obvious that this form of indirect addressing allows access to a 256 byte range of locations. In the case above, any location from $1500 and $15FF is available by setting the Y register accordingly. Program 17 uses post-indexed indirect addressing to move a line of screen memory from the top to the bottom of the screen.

Program 17

```
 10  REM ** INDIRECT ADDRESSING **
 20  CODE = 768
 30  FOR LOOP = 0 TO 9
 40      READ BYTE
 50      POKE CODE + LOOP, BYTE
 60  NEXT LOOP
 70  :
 80  REM ** M/C DATA **
 90  DATA 160,40      : REM $A0, $28      —LDY #$27
100  DATA 177,251     : REM $B1, $FB      —LDA ($FB) Y
```

```
110  DATA 145,253      :  REM $91, $FD      -STA ($FD), Y
120  DATA 136          :  REM $88           -DEY
130  DATA 208,249      :  REM $D0, $F9      -BNE -7
140  DATA 96           :  REM $60           -RTS
150  :
160  POKE 251,0 : POKE 252,4 : REM SCREENTOP
170  POKE 253,209 : POKE 254,7 : REM SCREENBOTTOM
180  HOME
190  PRINT " INDIRECT INDEXED ADDRESSING"
200  GET A$
210  IF A$ = " " THEN GOTO 200
220  CALL CODE
```

The program commences by assembling the machine code held in the data statements. Next, two vectors are created in zero page. The first (line 160) is POKEd with the address of the top left-hand corner of the screen (1024) and is called SCREENTOP. Similarly, in line 170, the next two locations are seeded to point to SCREENBOTTOM (2001).

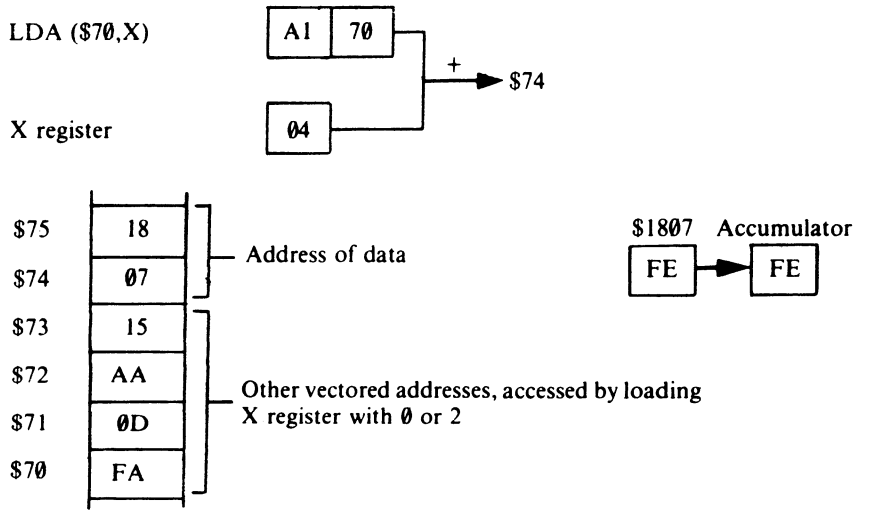
Once the program title has been printed and a key has been pressed, the machine code stored in the 'free' RAM area is executed. The Y register is set to the text screen line length count (line 90), then using post-indexed indirect addressing, the byte stored at SCREENTOP + Y is loaded into the accumulator (line 100) and stored in screen memory at the location specified by SCREENBOTTOM + Y (line 110). The Y register is decremented, thus allowing the next location to be accessed (line 120), and the process repeated until the Y register holds 0 (the BNE instruction in line 130 takes care of this, as we shall see in the next chapter).

The character codes for the title at the top of the screen are now stored in the last line displayed on the screen.

PRE-INDEXED ABSOLUTE ADDRESSING

This addressing mode is used if we wish to indirectly access a whole series of absolute addresses which are stored in zero page.

Operation:



Here the contents of the X register ($04) are added to the zero page address ($70) to give the vector address ($74). The two bytes here are then interpreted as the actual address of the data ($1807).

Setting the X register to $02 gives indirect access to the vector address $15AA. A list of instructions which can be used with pre- and post-indexed addressing is shown in Table 11.4.

Table 11.4

Pre- and post-indexed indirect addressing instructions

ADC	Add with carry
AND	Logical AND
CMP	Compare memory
EOR	Logical exclusive OR
LDA	Load accumulator
ORA	Logical OR
SBC	Subtract with carry
STA	Store accumulator

IMPLIED AND RELATIVE ADDRESSING

Two other modes of addressing are available with the 6502 namely, implied addressing and relative addressing. We will be dealing with both of these addressing modes during the course of the next few chapters.

12 Stacks of Fun

THE STACK

The stack is perhaps one of the more difficult aspects of the 6502 to understand, however it is well worth the time you will spend mastering it as it lends itself to more efficient programming. Because of its importance the whole of *Page $01* (that is memory locations $ 100 through to $1 FF) is given over to its operation.

The stack is used as a temporary store for data and memory addresses that need to be remembered for use sometime later on during the program. For most purposes its operation is transparent to us. For example, when, during the course of a BASIC program a GOSUB is performed, the address of the next BASIC command or statement after it is placed onto the stack, so that the program knows where to return to on completion of the procedure. The process of placing values onto the stack is known as *pushing*, while retrieving the data is called *pulling*.The stack has one important feature which must be understood—it is a *last-in-first-out* (LIFO) structure. What this means is that the last byte pushed onto the stack must be the first byte pulled from it.

A useful analogy to draw here is that of a train yard. Consider a small railroad siding, onto which freight cars 1, 2 and 3 are pushed (see Figure 12. 1). The first freight car onto the line (freight car 1) is at the very end of the line, freight car 2, the second onto the line is in the middle, and the last freight car (freight car 3) is nearest the points.

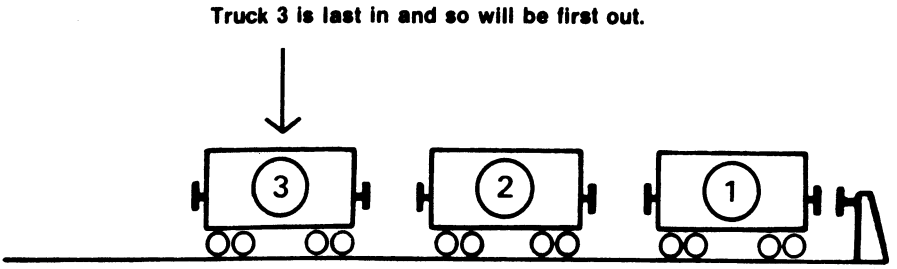


Figure 12.1 The stack—LIFO.

It should now be fairly obvious that the first freight car to be pulled off the railroad siding must be the last freight car pushed onto it, that is, freight car 3. Freight car 2 will be the next to be pulled from the line, and the first freight car in will be the last one out.

To help us keep track of our position on the stack, there is a further 6502 register called the *Stack Pointer*. Because the stack is a hardware item of the 6502, that is, it is actually 'wired' into it, the 'page number' of the stack ($01) can be omitted from the address, and the Stack Pointer just points to the next free position in the stack.

When the Apple // is switched on the Stack Pointer is loaded with the value $FF – it points to the top of the stack – this means that the stack grows down the memory map rather than up as may be expected. Each time an item is pushed the Stack Pointer is decremented, and conversely, it is incremented when the stack is pulled.

STACK INSTRUCTIONS FOR SAVING DATA

The 6502 has four instructions that allow the accumulator and Status register to be pushed and pulled. They are:

PHX	Push x-register onto stack
PHY	Push y-register onto stack
PHA	Push accumulator onto stack
PLA	Pull accumulator from stack
PHP	Push Status register onto stack
PLP	Pull Status register from stack
PLX	Pull x-register from stack
PLY	Pull y-register from stack

All four instructions use implied addressing and occupy only a single byte of memory.

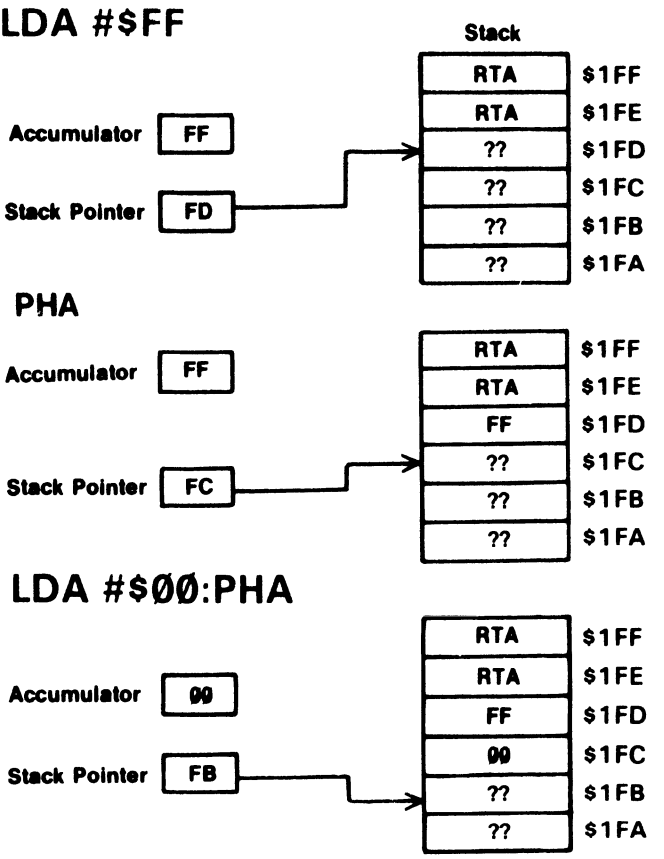


Figure 12.2 Pushing items on to the stack.

The PHA and PHP instructions work in a similar manner, but on different registers. In both cases the source register remains unaltered by the instruction. Again PLA and PLP are similar in operation, but PLA conditions only the Negative and Zero flags, while PLP of course conditions all the flags.

Consider the following sequence of instructions:

```
TWOPUSH  LDA #$FF        \  place $FF in accumulator
         PHA             \  push onto stack
         LDA #$00        \  place $00 in accumulator
         PHA             \  push onto stack
```

Figure 12.2 shows exactly what happens as this program is executed. The Stack Pointer (SP) at the start contains $FD and points to the next free location in the stack. The first two stack locations $FF and $FE hold the two byte return address (RTA) to which the machine code will eventually pass control. (This may be the address of the next BASIC instruction if a call to machine code has been made.) The subsequent stack locations are at present undefined and are therefore represented as ??.

After the accumulator has been loaded with $FF it is copied onto the stack by PHA. Note that the accumulator's contents are not affected by this operation. Once the value in the accumulator has been pushed onto the stack, the Stack Pointer's value is decremented by one to point to the next free location in the stack ($FC).

The accumulator is then loaded with $00 and this is pushed on to the stack at location $FC. The Stack Pointer is again decremented to the next free location ($FB).

To remove these items from the stack the following could be used:

```
TWOPULL  PLA             \  get $00 from stack
         STA TEMP        \  save it somewhere
         PLA             \  get $FF from stack
         STA TEMP + 1    \  save it as well
```

Figure 12.3 illustrates what happens in this case. The first PLA will pull from the stack into the accumulator the last item pushed onto it, which in this example is $00. The Stack Pointer is incremented, this time to point to the new 'next free location', $FC. As you can see from the diagram the stack contents are not altered, but the $00 wi11 be overwritten if a further item is now pushed. The STA TEMP saves the accumulator value somewhere in memory so that it is not destroyed by the next PLA. This PLA restores the value $FF into the accumulator, and again increments the Stack Pointer.

One thing should now be apparent—it is very important to remember the order in which items are pushed onto the stack, as they *must* be pulled off it in exactly the reverse order. If this process is not strictly adhered to then errors will certainly result, and could even cause your program to crash or hang-up!

The following program shows how the stack can be used to save the contents of the various registers to be printed later. This is particularly useful for debugging those awkward programs that just will not work.

```
REGSAVE   PHP      \  save Status register
          PHA      \  save accumulator
          TXA      \  transfer X into accumulator
          PHA      \  save accumulator (X)
          TYA      \  transfer Y into accumulator
          PHA      \  save accumulator (Y)
```

It is important to save the registers in the order shown. The Status register should be saved first so that it will not be altered by the subsequent transfer instructions which could affect the Negative and Zero flags, and the accumulator must be saved before its value is destroyed by either of the index register transfer operations.

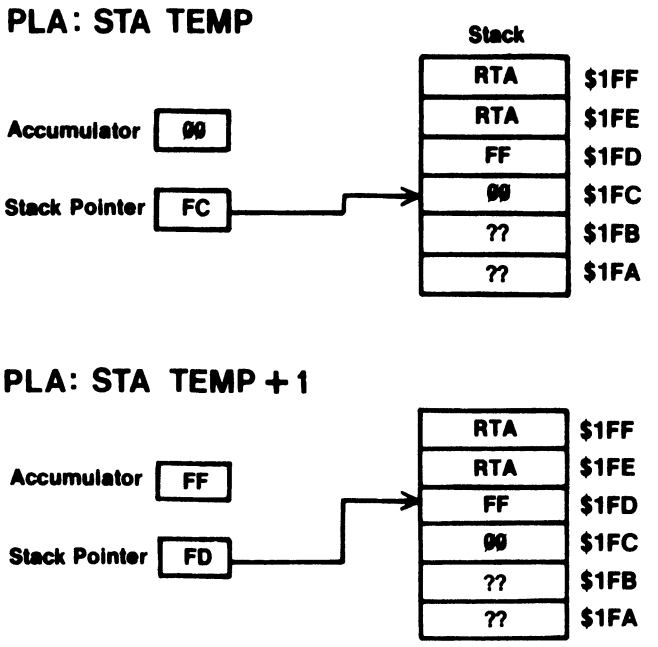


Figure 12.3 Pulling items from the stack.

If the registers' values had been saved to preserve them while another portion of the program was operating, we could retrieve them with:

```
PLA      \  pull accumulator (Y)
TAY      \  and transfer to Y register
PLA      \  pull accumulator (X)
TAX      \  and transfer to X register
PLA      \  pull accumulator
PLP      \  pull Status register
```

There are two final stack associated instructions:

```
TSX     Transfer Stack Pointer to X register
TXS     Transfer X register to Stack Pointer
```

These instructions allow the Stack Pointer to be seeded as required. On power-up the Apple does the following:

```
LDX #$FF                    \  load X with $FF
TXS                         \  place in Stack Pointer
```

It is very unlikely that you will ever need these two instructions unless you go on to such splendid projects as writing your own interpreter! We shall see how and why the stack is used to save addresses in Chapter 14.

13 Looping

LOOPS

Loops allow sections of programs to be repeated over and over again. For example, in BASIC we could print ten exclamation marks using a FOR... NEXT loop like this:

```
10    FOR NUMBER = 0 TO 9
20    PRINT "!" ;
30    NEXT NUMBER
```

In line 10 a counter called NUMBER is declared and initially set to zero. Line 20 prints the exclamation mark, and line 30 checks the present value of NUMBER to see if it has reached its maximum limit. If it has not, the program adds one to NUMBER and branches back to print the next exclamation mark.

To implement this type of loop in assembly language we need to know how to control and use the three topics identified above; namely counters, comparisons and branches.

COUNTERS

It is usual to use index registers as counters, because they have their own increment and decrement instructions.

INX	Increment X register	X = X + 1
INY	Increment Y register	Y = Y + 1
DEX	Decrement X register	X = X - 1
DEY	Decrement Y register	Y = Y - 1

All these instructions can affect the Negative and Zero flags. The Negative flag is set if the most significant bit of the register is set following an increment or decrement instruction – otherwise it will be cleared. The Zero flag will only be set if any of the instructions cause the register concerned to contain zero.

Note that incrementing a register which contains $FF will reset that register to $00, will clear the Negative flag (N = 0) and will set the Zero flag (Z = 1). Conversely,

decrementing a register holding $00 will reset its value to $FF, set the Negative flag and clear the Zero flag.

There are two other increment and decrement instructions:

INC	Increment memory
DEC	Decrement memory

These instructions allow the values in memory locations to be adjusted by one, for example:

```
INC $70                    \ add 1 to location $70
DEC $1500                  \ subtract 1 from location $1500
```

Both instructions condition the Negative and Zero flags as described earlier.

Program 18 shows how these instructions can be used, in this case to print 'ABC' on the screen.

Program 18

```
 10  REM ** INCREMENTING A REGISTER **
 20  CODE = 768
 30  FOR LOOP = 0 TO 16
 40      READ BYTE
 50      POKE CODE + LOOP, BYTE
 60  NEXT LOOP
 70  :
 80  REM ** M/C DATA **
 90  DATA 169,193     : REM $A9, $C1        -LDA #ASC"A"
100  DATA 170         : REM $AA             -TAX
110  DATA 232         : REM $E8             -INX
120  DATA 32,237,253  : REM $20, $ED, $FD   -JSR $FDED
130  DATA 138         : REM $8A             -TXA
140  DATA 232         : REM $E8             -INX
150  DATA 32,237,253  : REM $20, $ED, $FD   -JSR $FDED
160  DATA 138         : REM $8A             -TXA
170  DATA 32,237,253  : REM $20, $ED, $FD   -JSR $FDED
180  DATA 96          : REM $60             -RTS
190  :
200  CALL CODE
```

The meaning of each line is as follows:

Lines 20-60	Assemble machine code.
Line 90	Place ASCII 'A' in accumulator.
Line 100	Save it in X register.
Line 110	Increment X to give code for 'B'.
Line 120	Print 'A' to screen.
Line 130	Transfer ASCII code for 'B' to accumulator.
Line 140	Increment X to give code for 'C'.
Line 150	Print 'B' to screen.
Line 160	Transfer ASCII code for 'C' into accumulator.

Line 170 Print 'C' to screen.
Line 180 Back to BASIC.
Line 200 Execute machine code.

COMPARISONS

There are three compare instructions:

CMP	Compare accumulator
CPX	Compare X register
CPY	Compare Y register

The contents of any register can be compared with the contents of a specified memory location, or as is often the case, the value immediately following the mnemonic. The values being tested remain unaltered. Depending on the result of the comparison, the Negative, Zero and Carry flags are conditioned. How are these flags conditioned? Well, the first thing the 6502 does is set the Carry flag (C = 1). It then subtracts the specified value from the contents of the register. If the value is less than, or equal to the register contents, the Carry flag remains set. If the two values are equal the Zero flag is also set. If the Carry flag has been cleared, it means that the value was greater than the register contents, and a borrow occurred during the subtraction. The Negative flag is generally (but not always) set when this occurs — this is only really valid for two's complement compares. Table 13.1 summarizes these tests.

Table 13.1

Test	Flags		
	C	Z	N
Register less than data	0	0	1
Register equal to data	1	1	0
Register greater than data	1	0	0

BRANCHES

Depending on the result of a comparison, the program will need either to branch back to repeat the loop, branch to another point in the program, orjust simply continue. This type of branching is called *conditional branching*, and eight instructions enable various conditions to be evaluated. The branch instructions are:

BNE	Branch if not equal	Z = 0
BEQ	Branch if equal	Z = 1
BCC	Branch if Carry clear	C = 0
BCS	Branch if Carry set	C = 1

BPL	Branch if plus	N = 0
BMI	Branch if minus	N = 1
BRA	Branch always	
BVC	Branch if overflow clear	V = 0
BVS	Branch if overflow set	V = 1

Let's now rewrite the BASIC program to print ten exclamation marks (see page 62) in assembly language.

Program 19

```
 10  REM ** 10 ! MARKS **
 20  CODE =768
 30  FOR LOOP = 0 TO 12
 40      READ BYTE
 50      POKE CODE + LOOP, BYTE
 60  NEXT LOOP
 70  :
 80  REM ** M/C DATA **
 90  DATA 162,0          : REM $A2, $00          -LDX #0
100  DATA 169,161        : REM $A9, $A1          -LDA #ASC"!"
110  DATA 32,237,253     : REM $20, $ED, $FD     -JSR $FDED
120  DATA 232            : REM $E8               -INX
130  DATA 224,10         : REM $E0, $0A          -CPX #10
140  DATA 208,248        : REM $D0, $F8          -BNE -8
150  DATA 96             : REM $60               -RTS
160  :
170  CALL CODE
```

Lines 90 and 100 initialize the X register and place the ASCII code for an exclamation mark into the accumulator. Line 110 uses the JSR instruction to print the accumulator's contents to the screen. The X register is incremented (line 120) and if not yet equal to 10 the BNE instruction (line 140) is performed and the program loops back to print another exclamation mark.

If you look closely at the program listing, more especially at line 140, you will notice that the BNE opcode is followed by a single byte, and not an address as you may have expected. This byte is known as the *displacement*, and this type of addressing is called *relative* addressing. The operand, in this case 248 ($F8), tells the processor that a backward branch of 8 bytes is required.

To distinguish branches backwards from branches forward you use *signed binary*. A negative value indicates a backward branch while a positive number indicates a forward branch. Obviously, it is important to know how to calculate these displacements—so let's try it.

Before sitting down in front of your Apple // it is always best to write out your machine code program on paper. While it is perfectly feasible to write it 'at the keyboard', this nearly always leads to problems caused by errors in the coding (and I speak from experience). To make it clear just where loops are branching to and from, you can use labels. Table 13.2 shows the layout for Program 19.

Table 13.2

Label	Mnemonics	Comments
START		Code begins
	LDX #Ø	Loop counter
	LDA #ASC "!"	ASCII code for '!'
LOOP		Branch destination
	JSR $FDED	Print'!'
	INX	X = X + 1
	CPX #1Ø	Is X = 1Ø yet?
	BNE LOOP	No, continue
	RTS	All done!

To calculate the branch displacement, just count the number of bytes from the displacement byte itself back to the label LOOP.

```
LOOP
        JSR $FDED     3 bytes
        INX           1 byte
        CPX #10       2 bytes
        BNE LOOP      2 bytes
```

This gives a total displacement of 8 bytes. Note that the relative displacement and the BNE opcode are included in the count because the Program Counter will be pointing to the instruction after the branch.

To convert this backward displacement into its signed binary form, you just calculate its two's complement value (see Chapter 3 if you need some refreshing on how to do this).

```
  0000 1000   (8)
  1111 0111
+          1
-----------
  1111 1000   (-8 = $F8)
```

Since all branch instructions are two bytes long, effective displacements of -126 bytes (-128 + 2) and +129 bytes (127 + 2) are possible. To make life easier, you'll be relieved to know that Appendix 6 contains a couple of tables from which you can read off displacement values directly.

To demonstrate the use of a forward branch enter and RUN Program 20 which displays a 'Y' if location $FB contains a '0' or an 'N' otherwise.

Program 20

```
10  REM ** FORWARD BRANCHING **
20  CODE =768
30  FOR LOOP = 0 TO 14
40      READ BYTE
50      POKE CODE + LOOP, BYTE
```

```
 60  NEXT LOOP
 70  :
 80  REM ** M/C DATA **
 90  DATA 165,251      : REM $A5, $FB        —LDA $FB
100  DATA 240,6        : REM $F0, $06        —BEQ ZERO
110  DATA 169,206      : REM $A9, $CE        —LDA #ASC"N"
120                      REM BACK
130  DATA 32,237,253   : REM $20, $ED, $FD   —JSR $FDED
140  DATA 96           : REM $60             —RTS
150                      REM ZERO
160  DATA 169,217      : REM $A9, $D9        —LDA #ASC"Y"
170  DATA 24           : REM $18             —CLC
180  DATA 144,247      : REM $90, $F7        —BCC BACK 190
200  CALL CODE
```

In this program I have used labels contained within REM statements to identify the jump addresses. This should help to make things clearer.

The machine code begins by loading the byte at location $FB into the accumulator. If the byte is zero this will automatically set the Zero flag, and the branch of line 100 will be executed (BEQ — branch if equal). Because this is a forward branch a positive value is used to indicate the displacement—in this instance 6 bytes forward. The accumulator is then loaded with the ASCII code for Y. If the contents of $FB are non-zero, then the BEQ fails and the accumulator is loaded with 'N'.

Lines 170 and 180 illustrate a technique known as *forced branching*. The Carry flag is cleared and a BCC (branch carry clear) executed—because we cleared the Carry flag beforehand we have forced the processor to jump to BACK.

Whenever a register is used as a loop counter, and only as a loop counter, it is better to write the machine code so that the register counts down rather than up. Why? Well, you may recall from Chapter 7 that when a register is decremented such that it holds zero the Zero flag is set. Using this principle Program 19 can be re-written as follows, so that the CPX #10 instruction is superfluous:

Program 21

```
 10  REM ** DOWN COUNT **
 20  CODE = 768
 30  FOR LOOP = 0 TO 10
 40      READ BYTE
 50      POKE CODE + LOOP, BYTE
 60  NEXT LOOP
 70  :
 80  REM ** M/C DATA **
 90  DATA 162,10       : REM $A2, $0A        —LDX #10
100  DATA 169,161      : REM $A9, $A1        —LDA #ASC"!"
110                      REM LOOP
120  DATA 32,237,253   : REM $20, $ED, $FD   —JSR $FDED
130  DATA 202          : REM $CA             —DEX
140  DATA 208,250      : REM $D0, $FA        —BNE LOOP
150  DATA 96           : REM $60             —RTS
160  :
170  CALL CODE
```

FOR...NEXT

In BASIC the FOR...NEXT loop makes the Apple // execute a set of statements a specified number of times. All FOR...NEXT loops – including those containing positive and negative STEP sizes – are relatively easy to produce in machine code, and each type is summarized below.

1. FOR LOOP = FIRST TO SECOND...NEXT

This loop requires only two variables, the start and end values normally termed the entry and exit conditions. Here they are defined by the two variables FIRST and SECOND. No STEP size is indicated therefore the loop will increment by one each time round. In assembler this is implemented as:

```
SETUP   LDX FIRST       \  place loop start into counter
LOOP                    \  mark loop entry
        ...             \  loop statements here
        INX             \  add one to counter
        CPX SECOND      \  has loop limit been reached?
        BNE LOOP        \  no, execute loop again
```

2. FOR LOOP = SECOND TO FIRST STEP -1...NEXT

This loop is essentially the same as the previous one, except that the counter must initially be loaded with SECOND, and then decremented by one each time round to mimic the STEP -1 statement.

```
SETUP   LDX SECOND      \  place loop start in counter
LOOP                    \  mark loop entry
        ...             \  loop statements here
        DEX             \  decrement counter
        CPX FIRST       \  finished?
        BCS LOOP        \  no, execute again
```

In this loop the Carry flag remains set until the loop count is decremented below FIRST. Therefore, if SECOND = 10 and FIRST = 6 the loop is executed 5 times, just as it would be in BASIC.

3. FOR LOOP = FIRST TO SECOND STEP 3...NEXT

This loop is similar to that described in 1, except that three INX instructions are required to produce the STEP 3.

```
SETUP   LDX FIRST       \  place loop start in counter
LOOP                    \  loop entry
        ...             \  execute loop statements here
        INX             \  increment counter by 3
        INX
```

```
        CPX SECOND          \  finished?
        BNE LOOP            \  no, go again
```

On reaching SECOND the CPX instruction will succeed setting the Zero flag. The BNE LOOP will fail and the loop is completed.

4. FOR LOOP = SECOND TO FIRST STEP -3...NEXT

This loop is similar to that already described in 2, but needs three DEX instructions to generate the STEP -3.

```
SETUP   LDX SECOND          \  place loop start in counter
LOOP                        \  loop entry
        ...                 \  execute loop statements
        DEX                 \  decrement by three
        DEX
        DEX
        CPX FIRST           \  finished?
        BCS LOOP            \  no, go again
```

5. FOR LOOP = FIRST TO SECOND STEP NUM...NEXT

If NUM is known at the time of writing the program then just include the correct number of INX statements. However, if NUM = 10, ten INX instructions would be a pretty inefficient piece of programming. The rule here is to use INX for STEPs of 4 or less and otherwise to use the ADC instruction.

```
SETUP   LDX FIRST
LOOP
        ...
        PHA             \  save accumulator if needed
        TXA             \  move counter across into
                           accumulator
        CLC             \  clear Carry flag
        ADC NUM         \  add STEP size
        TAX             \  restore counter
        PLA             \  and accumulator
        CPX SECOND      \  finished?
        BNE LOOP        \  no, go again
```

Note here that the counter's contents must be transferred to the accumulator for the ADC NUM instruction to be performed, and returned to the X register on completion. If the accumulator's contents are important they can be preserved on the stack.

6. FOR LOOP = SECOND TO FIRST STEP -NUM...NEXT

The rules for 5 apply here, except that the Carry flag must first be set and SBC used to mimic the minus STEP size.

```
SETUP   LDX SECOND
LOOP
        ...
        PHA             \  save accumulator if needed
        TXA             \  move counter across
        SEC             \  get Carry flag
        SBC NUM         \  minus STEP
        TAX             \  restore counter
        PLA             \  and accumulator
        CPX FIRST       \  finished?
        BCS LOOP        \  no, go again
```

Of course, the Y register could have been used equally well as the loop counter.

MEMORY COUNTERS

Invariably programs that operate on absolute addresses will require routines that are capable of incrementing or decrementing these double byte values. A typical case being a program using post-indexed indirect addressing that needs to sequentially access a whole range of consecutive memory locations. The following two programs show how this can be done. First, incrementing memory addresses.

Program 22

```
 10  REM ** INCREMENTING MEMORY **
 20  CODE = 768
 30  FOR LOOP = 0 TO 6
 40      READ BYTE
 50      POKE CODE + LOOP, BYTE
 60  NEXT LOOP
 70  :
 80  REM ** M/C DATA **
 90  DATA 230,251     : REM $E6, $FB      - INC $FB
100  DATA 208,2       : REM $D0, $02      - BNE OVER
110  DATA 230,252     : REM $E6, $FC      - INC $FC
120                     REM OVER
130  DATA 96          : REM $60           - RTS
140  :
150  POKE 251,0 : POKE 252,0
160  CALL CODE
170  LOW = PEEK (251)
180  HIGH = PEEK (252)
190  NUM = HIGH * 256 + LOW
200  PRINT NUM
210  GOTO 160
```

Lines 90 to 120 contain the relevant code. Each time it is executed by the CALL CODE command (line 160) the low byte of the counter at $FB is incremented. When the low byte changes from $FF to $00 the Zero flag is set, and so the branch in line 100 will not take place allowing the high byte of the counter at $FC to be incremented. Decrementing a counter is a little less straight forward.

Program 23

```
 10  REM ** DECREMENTING MEMORY **
 20  CODE = 768
 30  FOR LOOP = 0 TO 8
 40      READ BYTE
 50      POKE CODE + LOOP, BYTE
 60  NEXT LOOP
 70  :
 80  REM ** M/C DATA **
 90  DATA 165,251     : REM $A5, $FB      -LDA $FB
100  DATA 208,2       : REM $D0, $02      -BNE LSBDEC
110  DATA 198,252     : REM $C6, $FC      -DEC $FC
120                     REM LSBDEC
130  DATA 198,251     : REM $C6, $FB      -DEC $FB
140  DATA 96          : REM $60           -RTS
150  :
160  POKE 251,0 : POKE 252,0
170  CALL CODE
180  LOW = PEEK (251)
190  HIGH = PEEK (252)
200  NUM = HIGH * 256 + LOW
210  PRINT NUM
222  GOTO 170
```

First, the accumulator is loaded with the low byte of the counter, $FB (line 90); this procedure will condition the Zero flag. If it is set, the low byte of the counter must contain $00, and therefore the high byte needs to be decremented (line 110)— the low byte of the counter will always be decremented (line 130).If all registers are being used the following alternative can be employed:

```
        INC COUNTER
        DEC COUNTER
        BNE LSBDEC
        DEC COUNTER + 1
LSBDEC  DEC COUNTER
```

The process of first incrementing and then decrementing the low byte of COUNTER will condition the Zero flag in the same manner as a load instruction.

14 Subroutines and Jumps

SUBROUTINES

If you are familiar with BASIC's GOSUB and RETURN statements you should have little difficulty understanding the two assembler equivalents:

JSR	Jump save return
RTS	Return from subroutine

If you are not familiar, I will explain.

Quite often during the course of writing a program you will find that a specific operation must be performed more than once, perhaps several times. Rather than typing in the same group of mnemonics on every occasion, which is both time consuming and increases the programs' length, they can be entered once, out of the way of the main program flow, and called when required. Not every piece of repetitive assembler warrants being coded into a subroutine, however. For example:

```
INX : DEY : STA Temp
```

is quite common (or something very similar). However, when assembled it only occupies four bytes of memory, which is the same memory requirement as a JSR...RTS call. Nothing is to be gained by introducing a subroutine here then, in fact, it will actually slow the program operation down by a few millionths of a second! On the other hand:

```
CLC : LDA Temp : ADC Value : STA Somewhere
```

might well warrant its own subroutine call as, if absolute addressing is employed, it may be up to ten bytes in length. Let's now look at a short program, which contains several subroutine calls to the Apple //'s monitor ROM.

Program 24

```
10  REM ** SUBROUTINE DEMO **
20  CODE = 768
30  FOR LOOP = 0 TO 12
40      READ BYTE
50      POKE CODE + LOOP, BYTE
60  NEXT LOOP
70  :
```

```
 80   REM ** M/C DATA **
 90   :
100   DATA 32,12,253     : REM $20, $0C, $FD    — JSR $FD0C
120   DATA 133,251       : REM $85, $FB         — STA $FB
130   DATA 230,251       : REM $E6, $FB         — INC $FB
140   DATA 165,251       : REM $A5, $FB         — LDA $FB
150   DATA 32,237,253    : REM $20, $ED, $FD    — JSR $FDED
160   DATA 96            : REM $60              — RTS
170   :
180   CALL CODE
```

This program uses two subroutine calls. The first (line 100) is to the monitor ROM RDKEY subroutine at 64780 ($FD0C) which is, in effect, a keyboard scan routine that returns a detected key's ASCII value in the accumulator. If no keypress is detected the program stays at the RDKEY routine until a key is pressed. Once a key is pressed it's code is stored in location 251 ($FB) and is incremented (line 130) before being loaded back into the accumulator. Finally, the COUT subroutine, which we have used several times before, is called to print the accumulator's contents to the screen. RUN the program to see the effect. Try pressing the 'A' key — a 'B' should be printed!

Now that we have taken a general overview of subroutine calls and their functions it will be useful to see just how they manage to do what they do.

The two instructions JSR and RTS must perform three functions between them. Firstly, the current contents of the Program Counter must be saved so that control may be returned to the calling program at some stage. Secondly, the 6502 must be told to execute the subroutine once it arrives there. Finally, program control must be handed back to the calling program.

The JSR instruction performs the first two requirements. To save the return address it pushes the two byte contents of the Program Counter onto the stack. The Program Counter at this stage will hold the address of the location containing the third byte of the three which constitute the JSR instruction. After pushing the Program Counter onto the stack, the operand specified by JSR is placed into the Program Counter, which effectively transfers control to the subroutine.

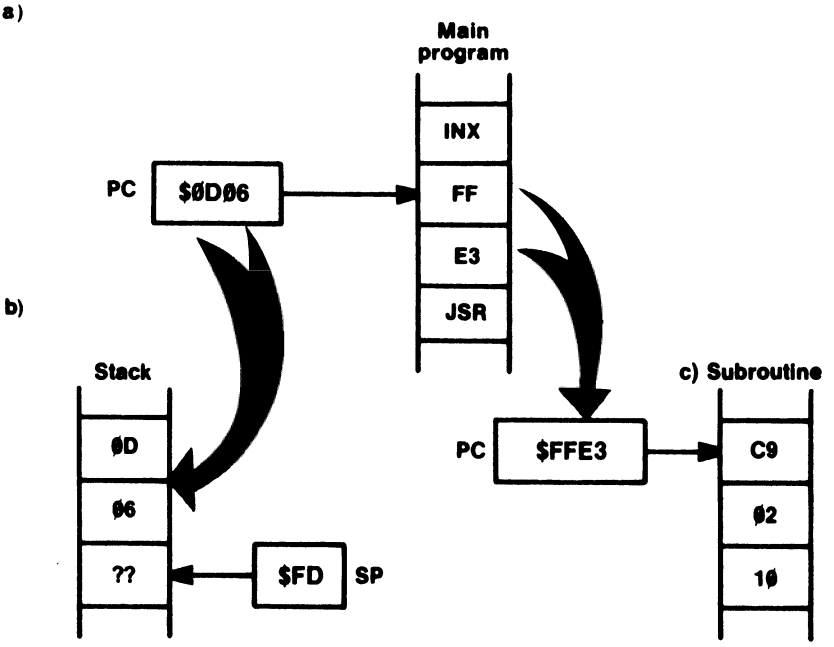


Figure 14.1 Steps taken by a JSR instruction.

Figure 14.1 shows how these operations take place, and in particular, their effect on the stack. At the time the 6502 encounters the JSR instruction the Program Counter is pointing to the second byte of the two byte operand (Figure 14.1a). The microprocessor pushes the contents of the Program Counter onto the stack, low byte first (Figure 14.1b), and then copies the subroutine address into the Program Counter (Figure I4.1c).

When the RTS instruction is encountered at the end of the subroutine, these actions are reversed. The return address is pulled from the stack and incremented by one (Figure 14.2) as it is replaced into the Program Counter, so that it points to the instruction after the original subroutine call.

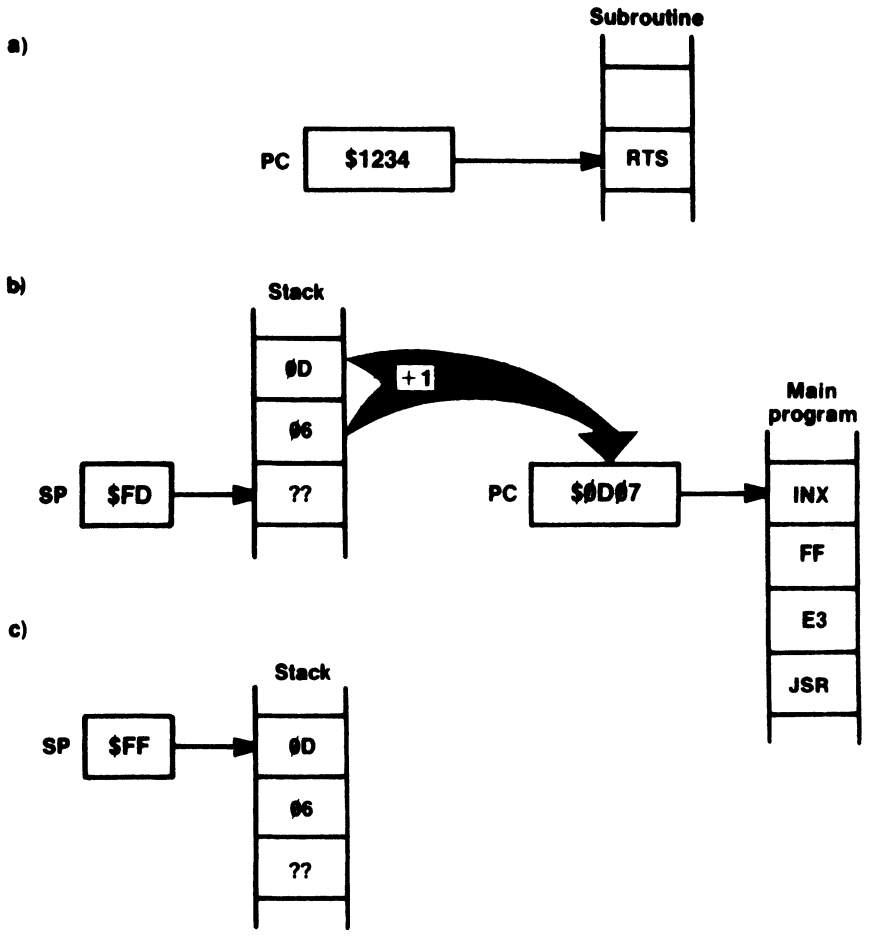


Figure 14.2 Steps taken by an RTS instruction.

PASSING PARAMETERS

Nine times out of ten a subroutine will require some data to work on, and this will have to be passed into the subroutine by the main program. For example, in Program 24, the values to be printed were placed into the accumulator before calling the COUT routine. This routine is written such that it expects to find a data byte in the accumulator.

Other subroutines may require several bytes of information, in which case the accumulator alone would not be sufficient. There are three general ways in which information or parameters can be passed into subroutines, these are:

1. Through registers.
2. Through memory locations.
3. Through the stack.

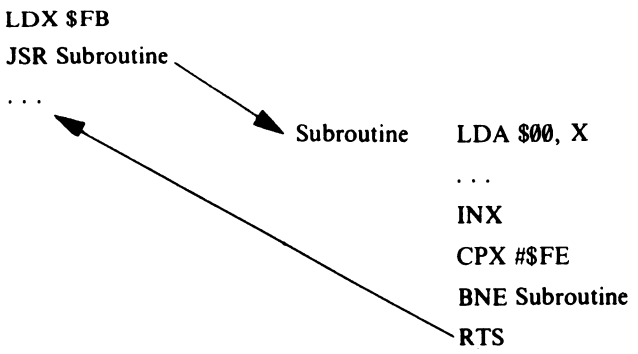
Let's look at each of these methods in turn.

Through registers

This is quite obviously the simplest method particularly because it can keep the subroutine independent of memory. Because only three registers are available though, only three bytes of information can be conveyed. The registers may themselves contain vital information, so this would need to be saved, possibly on the stack, for future restoration.

Through memory

This is probably the easiest method if numerous bytes are being passed into the subroutine. The most efficient way is to use the available memory locations on Zero page. If the subroutine uses several bytes of memory, a neat way of accessing them is to place the start address of the data in the X register, and then use zero page indexed addressing with $00 as the operand as follows:



The disadvantage of using memory locations, other than those on Zero page, to pass parameters is that it ties the subroutine to a given area, making it memory dependent. However, on most occasions this does not really matter.

Through the stack

Passing parameters through the stack needs care, since the top of the stack will contain the return address. This method also requires two bytes of memory in which the return address can be saved after pulling it from the stack (though, of course, the index registers could be used). If the stack is used, the subroutine needs to commence with:

```
PLA                 \ pull low byte
STA ADDR            \ and save it
PLA                 \ pull high byte
STA ADDR + 1        \ and save it
```

The STA instructions can be replaced by TAX and TAY respectively. It is common practice when using the index registers to hold an address, to place the low byte in the X register and the high byte in the Y register. Once the parameters have been pulled from the stack the return address can be pushed back on to it with,

```
LDA ADDR + 1
PHA
LDA ADDR
PHA
```

Remember, the stack is a LIFO structure, so the bytes need to be accessed and pushed in the reverse order from that in which they were pulled and saved.

If a variable number of parameters is being passed into the subroutine, the actual number can be ascertained each time by evaluating the contents of the Stack Pointer. This can be carried out by transferring its value to the X register with TSX, and incrementing the X register each time the stack is pulled until, say, $FF is reached, indicating the stack is empty. The actual value tested for will depend on whether any other subroutine calls were performed previously—making the current one a nested subroutine. The value $FF is therefore just a hypothetical case and assumes nothing other than that data is present on the stack.

JUMPS

The JMP instruction operates in a similar manner to BASIC's GOTO statement in that it transfers control to another part of the program. In assembly language however, an absolute address is specified rather than a line number (which does not, of course, exist in machine code). The instruction operates simply by placing the two byte address specified after the opcode into the Program Counter, effectively producing a jump.

Program 25 creates a continuous loop by jumping back to the start of the program. This is seen as an unending stream of asterisks being printed to the screen—you'll have to press RESET TO get back to BASIC.

Program 25

```
 10  REM ** JUMPING **
 20  CODE = 768
 30  FOR LOOP = 0 TO 7
 40      READ BYTE
 50      POKE CODE + LOOP, BYTE
 60  NEXT LOOP
 70  :
 80  REM ** M/C DATA **
 90  REM START
100  DATA 169,170      : REM $A9, $AA       —LDA #ASC"*"
110  DATA 32,237,253   : REM $20, $ED, $FD  —JSR $FDED
120  DATA 76,0,3       : REM $4C, $00, $03  —JMP START
130  :
140  CALL CODE
```

JMP will generally be used to leapfrog over a section of machine code that need not be executed because a test failed. For example:

```
            BCC OVER
            JMP SOMEWHERE
OVER        LDA BYTE
            ASL A
            INX
            DEY
SOMEWHERE   STA TEMP
```

Here, if the Carry flag is clear, the jump instruction will be skipped and the code of OVER executed. If the Carry flag is set the test will fail, the JMP will be encountered and the code of OVER bypassed. A further use of JMP, the 'indirect jump', was detailed in Chapter 11. As we saw, the address that is actually jumped to is stored in a vector, the address of which is specified in the instruction. JMP ($FB) being an example.

15 Shifts and Rotates

Basically, these instructions allow the bits in a single byte to be moved one bit to the left or one bit to the right. There are four instructions available:

ASL	Arithmetic shift left
LSR	Logical shift right
ROL	Rotate left
ROR	Rotate right

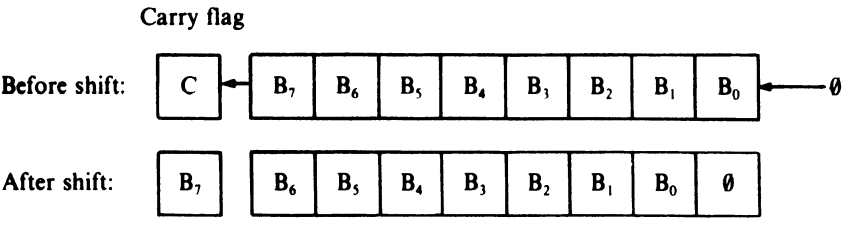
All of these instructions may operate directly on the accumulator, or on a specified memory byte:

```
ASL A          \ arithmetic shift left accumulator
ROL $FB        \ rotate left location $FB
```

Let's investigate each command in more detail.

ARITHMETIC SHIFT LEFT

ASL moves the contents of the specified byte left by a single bit.



Bit 7 (B_7) is shifted into the Carry flag, and a '0' takes the place of bit 0 as the rest of the bits are shuffled left. The overall effect is to double the value of the byte in question.

Example:

```
                                                    C       Accumulator
LDA $42          \   load accumulator with $42     X       0 1 0 0 0 0 1 0

                                                    C       Accumulator
ASL A            \   shift accumulator left        0       1 0 0 0 0 1 0 0
```

The accumulator now holds $84, twice the original value!

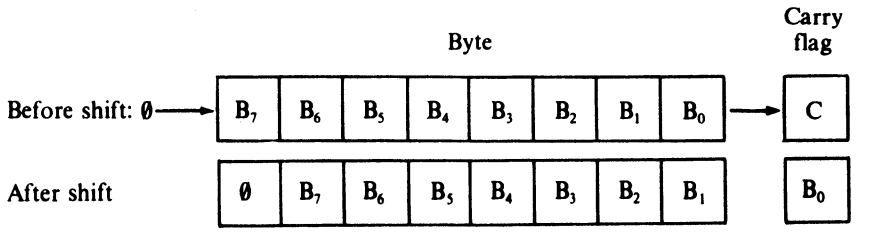
A further example of ASL is given by Program 26 which asks for a number (less than 64), multiplies it by four using ASL A, ASL A, and prints the answer.

Program 26

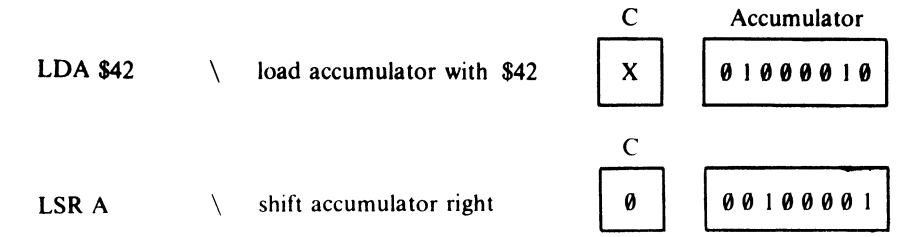
```
 10  REM ** MULTIPLY BY FOUR **
 20  CODE = 768
 30  FOR LOOP = 0 TO 6
 40      READ BYTE
 50      POKE CODE + LOOP, BYTE
 60  NEXT LOOP
 70  :
 80  REM ** M/C DATA **
 90  DATA 165,251     : REM $A5, $FB      — LDA $FB
100  DATA 10          : REM $0A           — ASL A
110  DATA 10          : REM $0A           — ASL A
120  DATA 133,251     : REM $85, $FB      — STA $FB
130  DATA 96          : REM $60           — RTS
140  :
150  HOME
160  INPUT "NUMBER TO BE MULTIPLIED BY 4: "; NUM
170  POKE 251,NUM
180  PRINT NUM; " x 4= ";
190  CALL CODE
200  PRINT PEEK (251)
```

LOGICAL SHIFT RIGHT

LSR is similar to ASL except that it moves the bits in the opposite direction, with bit 0(B_0) jumping into the Carry flag and a 0 following into the spot vacated by bit 7 (B_7).



This instruction could well have been called arithmetic shift right because it effectively divides the byte being shifted by two. For example:



The accumulator now holds $21, half the original value.
Using:

```
LSR A : BCS Elsewhere
```

or,

```
LSR A : BCC Somewhere
```

is a good efficient way of testing bit 0 of the accumulator.

Program 27 tests the condition of bit 0 of an input ASCII character by shifting it into the Carry flag position. If the carry is clear a zero is printed, if set—a one is printed instead.

Program 27

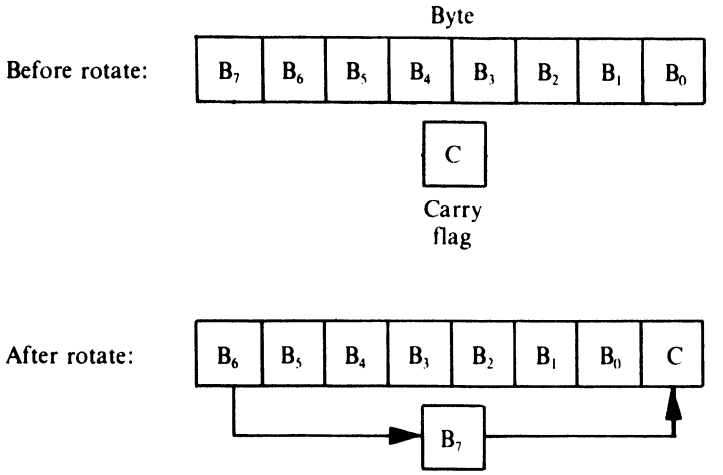
```
 10  REM ** TEST BIT 0 **
 20  CODE = 768
 30  FOR LOOP = 0 TO 10
 40      READ BYTE
 50      POKE CODE + LOOP, BYTE
 60  NEXT LOOP
 70  :
 80  REM ** M/C DATA **
 90  DATA 165,251     : REM $A5, $FB       —LDA $FB
100  DATA 74          : REM $4A            —LSR A
110  DATA 169,176     : REM $A9, $B0       —LDA #ASC"0"
120  DATA 105,0       : REM $69, $00       —ADC #0
130  DATA 32,237,253  : REM $20, $ED, $FD  —JSR $FDED
140  DATA 96          : REM $60            —RTS
150  :
160  INPUT A
170  POKE 251, A
180  CALL CODE
```

The input value (line 160) is POKEd into location 251 ($FB). This is then loaded into the accumulator (line 90) and a logical shift right performed (line 100) which moves bit 0 into the Carry flag position—either setting or clearing it. The accumulator is then loaded with the ASCII code value for '0' (Line 120) before the ADC instruction adds 0 to it (line 130)! No that's not as crazy as it seems—remember that the ADC

instruction takes the value of the Carry flag into consideration. If it is clear then the accumulator will still hold the ASCII code for 0, but if it is set, one will be added to the accumulator's value so that it now holds the ASCII code for 1!

ROTATE LEFT

This instruction uses the Carry flag as a ninth bit, rotating the whole byte left one bit in a circular motion, with bit 7 moving into the Carry flag, which in turn moves across to bit 0.



ROL provides an easy method of testing any of the four bits constituting the upper nibble of the accumulator. The desired bit is rotated into the bit 7 position, thus setting or clearing the Negative flag as appropriate.

Example: test bit 5 of accumulator.

Accumulator AF 1 0 1 0 1 1 1 1 N = 1

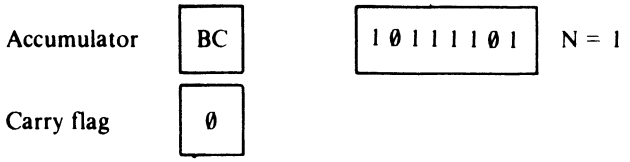
Carry flag 0

ROL A / rotate bit 6 into the bit 7 position

Accumulator 5E 0 1 0 1 1 1 1 0 N = 0

Carry flag 1

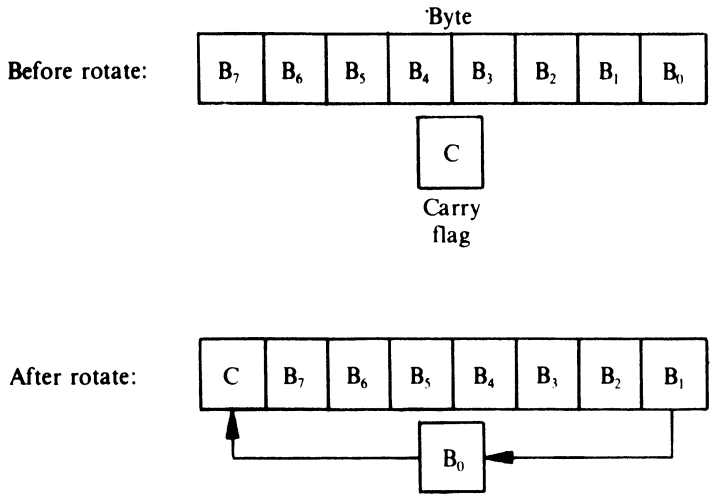
ROL A / rotate bit 5 into the bit 7 position



The Negative flag is now set, indicating that bit 5 of the accumulator was set.

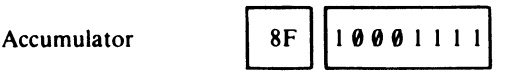
ROTATE RIGHT

Works just like ROL except the bits move to the right.

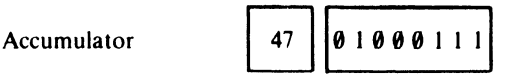


Example: ROR accumulator containing $8F.

```
CLC          \  clear Carry flag
LDA $8F      \  load accumulator with $8F
```




```
ROR          \  rotate right
```



If you need to shift (or rotate) the contents or a particular location several times, it is more efficient to load the value into the accumulator, shift (or rotate) that and store it back, than to manipulate the location directly.For example, to rotate location $1234 to the right four times, we could use:

```
ROR $1234
ROR $1234
ROR $1234
ROR $1234
```

This uses twelve bytes of memory, four for the instructions and eight for addresses. Alternatively:

```
LDA $1234
ROR A
ROR A
ROR A
ROR A
STA $1234
```

uses two bytes less and is 25% quicker in operation.

So far we have only considered shifting and rotating single bytes. By using combinations of instructions it is possible to perform similar operations on two byte values such as $CAFE.

To perform an overall ASL on two bytes located at HIGH and LOW, ASL and ROL are used in conjunction:

```
ASL HIGH        \ shift bit 7 by LOW into Carry flag
ROL LOW         \ rotate it into bit 0 of HIGH
```

By exchanging the commands an overall LSR on the same two bytes can be performed:

```
LSR HIGH        \ shift bit 0 HIGH into Carry flag
ROR LOW         \ rotate it into bit 7 of LOW
```

Note that the bytes are manipulated in the reverse order because we wish to move the bits in the opposite direction. As with single byte shifts, the two byte values can be doubled or divided in half.

Two byte rotates to move the bits in a circular manner, are simply rotation operations performed twice! However, as with two byte shifts, it is important to get the byte rotation order correct.

A two byte ROR is performed with:

```
ROR HIGH        \ rotate bit 0 of HIGH into Carry flag
ROR LOW         \ and on into bit 7 of LOW
```

While two byte ROLs are implemented with:

```
ROL LOW          \ rotate bit 7 of LOW into Carry flag
ROL HIGH         \ and on into bit 0 of HIGH
```

Finally, moving back to single byte shifts, to shift the contents of the accumulator right one bit while preserving the sign bit, use the following technique:

```
TAY              \ save accumulator in Y register
ASL A            \ move sign bit (bit 7) into Carry flag
TYA              \ restore original value back into
                   accumulator
ROR A            \ rotate right moving sign bit back
                   into bit 7
```

The Y register has been used as a temporary store for the accumulator. We could have used the X register or a memory location with equal effect.

PRINTING BINARY!

Quite often, it is necessary to know the binary bit pattern that a register or memory location holds. This is particularly true in the case of the Status register when the condition of its flags can often provide a great deal of information about the way a program is running.

Program 28 shows how the binary value of a byte can be printed. It uses the Status register's contents at the time the program is RUN as an example.

Program 28

```
 10  REM ** BINARY OUTPUT OF SR **
 20  CODE = 768
 30  FOR LOOP = 0 TO 18
 40      READ BYTE
 50      POKE CODE + LOOP, BYTE
 60  NEXT LOOP
 70  :
 80  REM ** M/C DATA **
 90  DATA 8           : REM $08            - PHP
100  DATA 104         : REM $68            - PLA
110  DATA 133,251     : REM $85, $FB       - STA $FB
120  DATA 162,8       : REM $A2, $08       - LDX #$08
130                     REM NBIT
140  DATA 6,251       : REM $06, $FB       - ASL $FB
150  DATA 169,176     : REM $A9, $B0       - LDA #ASC"0"
160  DATA 105,0       : REM $69, $00       - ADC #$00
170  DATA 32,237,253  : REM $20, $ED, $FD  - JSR $FDED
180  DATA 202         : REM $CA            - DEX
190  DATA 208,244     : REM $D0, $F4       - BNE NBIT
200  DATA 96          : REM $60            - RTS
210  :
```

```
220  HOME
230  PRINT "NV-BDIZC"
240  CALL CODE
```

The Status register needs to be saved in a memory location so that it can be manipulated. To do this it must be transferred into the accumulator by first pushing it on to the stack with PHP (line 90) and then pulling it into the accumulator with PLA (line 100); it can then be stored in zero page at location 251 (line 110). The X register is used to count the eight bits of the byte, so it is initialized accordingly (line 120). Line 130 is used as a label marker for NBIT (short for next bit). The arithmetic shift left (line 140) moves the msb of location 251 ($FB) into the Carry flag. The bit value (0 or 1) is printed out using the ASCII code for 0 and adding the Carry flag contents to it (lines 150 to 170) as described for Program 27. The bit counter is decremented (line 180) and a branch to NBIT executed if X has not reached zero (line 190).

To prove that the program does work, try including lines such as:

```
85  DATA 169,255   :  REM $A9, $FF    — LDA #$FF set N
85  DATA 169,0     :  REM $A9, $00    — LDA #$00 clear N set Z
```

These will condition the status flags as indicated. (Don't forget to change the LOOP count in line 30 from 18 to 20 when you add line 85.) You might like to try modifying the program to print the binary value of any key pressed on the keyboard!

BIT

The instruction, BIT, allows individual bits of a specified memory location to be tested. It has an important feature in that it does not change the contents of either the accumulator or the memory location being tested, but, as you may have guessed, it conditions various flags within the Status register. Thus:

1. The Negative flag is loaded with the value of bit 7 of the location being tested.
2. The Overflow flag is loaded with the value of bit 6 of the location being tested.
3. The Zero flag is set if the AND operation between the accumulator and the memory location produces a zero.

By loading the accumulator with a mask it is possible to test any particular bit of a memory location. For example, to test location TEMP to see if bit 0 is clear the following could be used:

```
LDA #1          \  000000011
BIT TEMP        \  test bit 0
```

If bit 0 of TEMP contains a 0, the Zero flag will be set, otherwise it will remain clear, thus allowing BNE and BEQ to be used for testing purposes.

This masking procedure need only be used for testing bits 0 to 5 because bits 6 and 7 are automatically copied into the Negative and Overflow flags, which have their own test instructions.

```
BIT TEMP
BMI                     \   branch if bit 7 set
BPL                     \   branch if bit 7 clear
BVC                     \   branch if bit 6 set
BVS                     \   branch if bit 6 clear 16
```

16 Multiplication and Division

MULTIPLICATION

Performing multiplication in assembly language is not too difficult provided that you have grasped what you have read so far. Unfortunately there are no multiplication instructions within the Apple //s 6502 (65C02) instruction set, therefore it is necessary to develop an algorithm to carry out this procedure.

Let's first look at the simplest method of multiplying two small values together. Consider the multiplication 5 x 6. We know the result is 30, but how did we obtain this? Simply by adding together six lots of five, in other words: 5 + 5 + 5 + 5 + 5 + 5 = 30. This is quite easy to implement:

Program 29

```
 10  REM ** SIMPLE MULTIPLICATION **
 20  CODE = 768
 30  FOR LOOP = 0 TO 16
 40      READ BYTE
 50      POKE CODE + LOOP, BYTE
 60  NEXT LOOP
 70  :
 80  REM ** M/C DATA **
 90  DATA 169,0      : REM $A9, $00      -LDA #$00
100  DATA 133,251    : REM $85, $FB      -STA $FB
110  DATA 162,6      : REM $A2, $06      -LDX #$06
120  DATA 24         : REM $18           -CLC
130                    REM LOOP
140  DATA 165,251    : REM $A5, $FB      -LDA $FB
150  DATA 105,5      : REM $69, $05      -ADC #$05
160  DATA 133,251    : REM $85, $FB      -STA $FB
170  DATA 202        : REM $CA           -DEX
180  DATA 208,247    : REM $D0, $F7      -BNE LOOP
190  DATA 96         : REM $60           -RTS
200  :               :
210  CALL CODE
220  PRINT "RESULT = ";
230  PRINT PEEK (251)
```

All we have done here is to create a loop to add 5 to the contents of location $FB six times to produce the desired result! This method is reasonable for multiplying small values, but not particularly efficient for larger numbers.

At this point, it might be worth reviewing the usual procedure for multiplying two large decimal numbers together. Consider 123 x 150. We would approach this, (without calculators, please!) thus:

```
  123     (Multiplicand)
 ×150     (Multiplier)
 ----
  000     (Partial product 1)
 615      (Partial product 2)
123       (Partial product 3)
-----
18450     (Result or final product.)
```

The initial two values are termed the multiplicand and multiplier, and their product is formed by multiplying, in turn, each digit in the multiplier by the multiplicand. This results in a partial product, which is written such that its least significant digit sits directly below the multiplier digit to which it corresponds. When formation of all the partial products is completed, they are added together to give the final product or result.

We can apply this technique to binary numbers, starting off with two three bit values, 010 X 011.

```
  010     (Multiplicand)
 ×011     (Multiplier)
 ----
  010     (Partial product 1)
 010      (Partial product 2)
000       (Partial product 3)
-----
00110     (Result)
```

Ignoring leading zeros, we obtain the result 110 (2 x 3 = 6). Moving on to our original decimal example, its binary equivalent is:

```
    01111011      123($7B)
   ×10010110      150($96)
   --------
    00000000
   01111011
  01111011
 00000000
01111011
00000000
01111011
--------
1001000000100110  18450($4812)
```

Hopefully you will have noticed that if the multiplier digit is a 0 it will result in the whole partial product being a line of zeros (anything multiplied by zero is zero). Therefore if a 0 is present in the multiplier it can simply be ignored but we must remember to shift the next partial product up past any 0s so that its least significant digit still corresponds to the correct 1 of the multiplier. This technique of shifting and ignoring can be used to write an efficient multiplication program.

Program 30

```
 10  REM ** SINGLE BYTE MULT GIVING 2 BYTE RESULT **
 20  CODE = 768
 30  FOR LOOP = 0 TO 19
 40      READ BYTE
 50      POKE CODE + LOOP, BYTE
 60  NEXT LOOP
 70  :
 80  REM ** M/C DATA **
 90  DATA 162,8        : REM $A2, $08      -LDX #$08
100  DATA 169,0        : REM $A9, $00      -LDA #$00
110                      REM AGAIN
120  DATA 70,252       : REM $46, $FC      -LSR $FC
130  DATA 144,3        : REM $90, $03      -BCC OVER
140  DATA 24           : REM $18           -CLC
150  DATA 101,251      : REM $65, $FB      -ADC $FB
160                      REM OVER
170  DATA 106          : REM $6A           -ROR A
180  DATA 102,253      : REM $66, $FD      -ROR $FD
190  DATA 202          : REM $CA           -DEX
200  DATA 208,243      : REM $D0, $F3      -BNE AGAIN
210  DATA 133,254      : REM $85, $FE      -STA $FE
220  DATA 96           : REM $60           -RTS
230  :
240  HOME
250  INPUT "MULTIPLICAND: "; A
260  INPUT "MULTIPLIER: "; B
270  POKE 251, A : POKE 252, B
280  CALL CODE
290  HIGH = PEEK (254) : LOW = PEEK (253)
300  RESULT = HIGH * 256 + LOW
310  PRINT "RESULT IS ";
320  PRINT RESULT
```

This program takes two single byte numbers, multiplies them together storing the result (which may be 16 bits long) in zero page. Unlike the binary multiplication examples, it does not compute each partial product before adding them together, but totals the partial products as they are evaluated. This is a somewhat quicker method, because the final product is generated as soon as the last bit of the multiplier has been examined.

When RUN the program requests you to enter the multiplicand and multiplier (lines 250 and 260); these single byte values are then POKEd into memory. The machine code begins by setting the X register to 8 ready to act as the bit counter (line 90). The accumulator is then cleared—this is important as it affects the high result value in location 254 ($FE).

The main loop is marked by the label in line 110. Bit 0 of the multiplier is then shifted into the Carry flag (line 120) and a branch to OVER executed if the Carry is clear (line 140). If set, the multiplicand value needs to be added to the accumulator (line 150). The product value now in the accumulator is rotated right (line 170) and a further ROR on the low result byte at 253 ($FD) performed (line 180). These two operations move bit 0 of the accumulator into bit 7 of the low result byte. The X register is decremented (line 190) and the procedure repeated for bits 1-7.

As may have become clear in the last binary example, the procedure is—if the bit is set then add the byte, else ignore it and move on to the next shifting operation.

DIVISION

When performing the division of one number by another, we are actually calculating the number of times the second number can be subtracted from the first. Consider 125 divided by 5:

```
                25      (Quotient)
              ┌────
(Divisor)   5 | 125     (Dividend)
             -10
              ──
               25
               25
               ──
                0      (Remainder)
```

Here, 5 can be subtracted from 10 twice, so we note the value 2 as part of the quotient. The 10 is brought down and subtracted from the first two digits of the dividend, leaving 2. Because 5 cannot be subtracted from 2 the remaining 5 of the dividend is brought down to give 25. 5 can be subtracted from this, without remainder, 5 times. Again this is recorded in the quotient, which now reflects the final result.

To divide binary numbers, this same procedure is pursued. The above example in binary would look like this:

```
          00011001
        ┌────────
0101 | 01111101
       0101
       ────
         10
         ──
         101
         101
         ───
           0
           ─
           101
           101
           ───
             0
```

In fact, as you may see, dividing binary numbers is much simpler than dividing decimal numbers. If the divisor is less than or equal to the dividend the corresponding bit in the quotient will be a 1. If the subtraction is not possible a 0 is placed in the quotient, the next bit of the dividend is brought down, and the procedure repeated.

The following utility program divides two single byte values and indicates whether a remainder is present:

Program 31

```
10   REM ** SINGLE BYTE DIVIDE **
20   CODE = 768
30   FOR LOOP = 0 TO 20
40        READ BYTE
50        POKE CODE + LOOP, BYTE
60   NEXT LOOP
70   :
80   REM ** M/C DATA **
90   DATA 162,8          :  REM $A2, $08          — LDX #$08
```

```
100  DATA 169,0        : REM $A9, $00      -LDA #$00
110                    REM AGAIN
120  DATA 6,251        : REM $06, $FB      -ASL $FB
130  DATA 42           : REM $2A           -ROL A
140  DATA 197,252      : REM $C5, $FC      -CMP $FC
150  DATA 144,4        : REM $90, $04      -BCC OVER
160  DATA 229, 252     : REM $E5, $FC      -SBC $FC
170  DATA 230,251      : REM $E6, $FB      -INC $FB
180                    REM OVER
190  DATA 202          : REM $CA           -DEX
200  DATA 208,242      : REM $D0, $F2      -BNE AGAIN
210  DATA 133,253      : REM $85, $FD      -STA $FD
220  DATA 96           : REM $60           -RTS
230  :
240  PRINT
250  INPUT "DIVIDEND: "; A
260  INPUT "DIVISOR: "; B
270  POKE 251,A : POKE 252,B
280  CALL CODE
290  PRINT "RESULT = ";
300  PRINT PEEK (251)
310  PRINT "REMAINDER = ";
320  PRINT PEEK (253)
```

In case you cannot readily follow what the program is doing, here is a line by line description of the mnemonics:

Line 90 Initialize the X register to indicate the number of bits to be shifted— 1 byte = 8 bits.
Line 100 Clear accumulator which will hold partial dividend value.
Line 110 Set loop.
Line 120 Shift the dividend left to provide the least significant bit position for the next digit of the quotient.
Line 130 The dividend bit is shifted left so that another bit from the partial dividend (which is in accumulator) can be tested.
Line 140 The divisor and partial dividend are compared.
Line 150 If the result indicates that the divisor is less than, or equal to the partial dividend...
Line 160 ...the divisor is subtracted from the partial dividend...
Line 170 ...and a 1 added to the quotient.
Line 180 If compare shows that the divisor is greater than the partial dividend, these last two lines are skipped.
Line 190 The bit count is decremented...
Line 200 ...and control returned to line 110 if not complete.
Line 210 Any remainder is saved in $FD.

This program uses the shift instructions of lines 120 and 130 as a two byte shift register in which the accumulator acts as the higher byte. The carry produced by ROL A is insignificant, in fact it is 0, and is eroded by the next ASL $FB procedure.

17 Assembly Types

CONDITIONAL ASSEMBLY

Conditional assembly allows different parts of an assembly language program to be assembled in response to certain different conditions being met. One of its main advantages is that a general source file can be created, rather than developing different source files for each particular need. The relevant code can be selected in response to a menu, or simply by seeding variable parameters.

Program 32 contains two different routines (multibyte addition and multibyte subtraction), but, depending on the response to the menu selection, only one of them will be assembled.

Program 32

```
 10  REM ** CONDITIONAL ASSEMBLY **
 20  CODE = 768
 30  HOME
 40  PRINT SPC(8)
 50  PRINT "CONDITIONAL ASSEMBLY"
 60  PRINT "PRESS 1 TO ASSEMBLE MBADD"
 70  PRINT " 2 TO ASSEMBLE MBSUB"
 80  GET A$
 90  A = VAL(A$)
100  IF A = 1 THEN GOTO 200
110  IF A = 2 THEN GOTO 300
120  GOTO 80
130  :
190  REM ** ASSEMBLE MBADD **
200  RESTORE
210  FOR LOOP = 0 TO 18
220       READ BYTE
230       POKE CODE + LOOP, BYTE
240  NEXT LOOP
250  END
260  :
290  REM ** ASSEMBLE MBSUB **
300  RESTORE
310  READ BYTE
320  IF BYTE < > 96 THEN GOTO 310
```

```
330  GOTO 210
340  :
350  REM ** MBADD DATA **
360  DATA 164,251    : REM $A4, $FB      -LDY $FB
370  DATA 162,0      : REM $A2, $00      -LDX #$00
380  DATA 24         : REM $18           -CLC
390                    REM AGAIN
400  DATA 189,0,21   : REM $BD, $00, $15 -LDA $1500,X
410  DATA 125,0,22   : REM $7D, $00, $16 -ADC $1600,X
420  DATA 157,0,21   : REM $9D, $00, $15 -STA $1500,X
430  DATA 232        : REM $E8           -INX
440  DATA 136        : REM $88           -DEY
450  DATA 208,243    : REM $D0, $F3      -BNE AGAIN
460  DATA 96         : REM $60           -RTS
470  :
480  REM ** MBSUB DATA **
490  DATA 164,251    : REM $A4, $FB      -LDA $FB
500  DATA 162,0      : REM $A2, $00      -LDX #$00
510  DATA 56         : REM $38           -SEC
520                    REM AGAIN
530  DATA 189,0,23   : REM $BD, $00, $17 -LDA $1700,X
540  DATA 253,0,24   : REM $FD, $00, $18 -SBC $1800,X
550  DATA 157,0,23   : REM $9D, $00, $17 -STA $1700,X
560  DATA 232        : REM $E8           -INX
570  DATA 136        : REM $88           -DEY
580  DATA 208,243    : REM $D0, $F3      -BNE AGAIN
590  DATA 96         : REM $60           -RTS
```

If item 1 is selected from the menu then the machine code is read and assembled in the normal manner by the BASIC loop of lines 210 to 240. If item 2 is selected we need to move the DATA pointer on so that it points to the first data byte in the MBSUB data listing. To do this a dummy READing loop is initiated in lines 310-320. This loop keeps reading bytes until BYTE = 96. This marks the end of the MBADD routine as 96 is of course the opcode for the RTS instruction. Now that the DATA pointer is in the correct position line 330 loops back for the machine code of MBSUB to be assembled in the normal manner.

MBADD uses absolute indexed addressing to sequentially fetch a byte of one operand (line 400) and add it to the corresponding byte of the other operand (line 410). The result is then stored in the location of the first operand. The Y register, having previously been loaded with the number of addition bytes via the zero page location 251 ($FB), is decremented (line 440), and the operation is continued until the Y register value has been exhuasted. The two multibyte numbers for addition should, of course, be stored in memory from locations $1500 and $1600 respectively.

The second routine, lines 490 to 590, is a multibyte subtraction routine (MBSUB) and is assembled by selecting item 2 on the menu. MBSUB also uses absolute indexed addressing to subtract the multibyte number at $1800 onwards from a similar number at $1700 onwards. Again the byte length of the number is stored in 251 ($FB).

LOOK-UP TABLES

Look-up tables provide a neat, compact and efficient way of obtaining data for what might otherwise turn out to be long and complicated machine code programs. For example, suppose we want to develop a machine code program to convert degrees Centigrade into degrees Fahrenheit. The formula for this is:

$°F = 1.8 (°C) + 32$

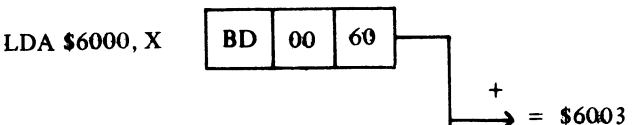
As you can see, this requires two mathematical operations—first a multiplication then an addition (a bit painful to the grey matter!). By providing the conversion values precalculated in a table, the Fahrenheit values can be extracted by using the Centigrade value as in index to the table. Try the following program:

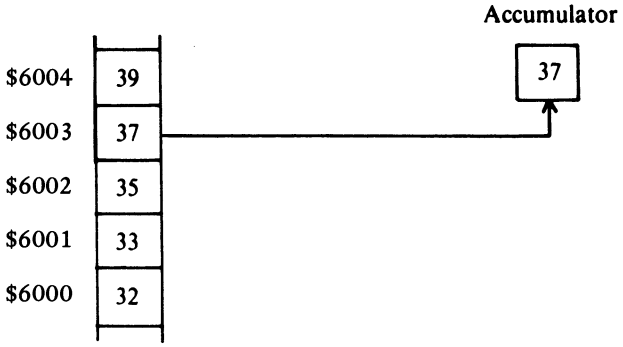
Program 35

```
 10  REM ** CENTIGRADE TO FAHRENHEIT **
 20  CODE = 768
 30  FOR LOOP = 0 TO 7
 40      READ BYTE
 50      POKE CODE + LOOP, BYTE
 60  NEXT LOOP
 70  :
 80  REM ** M/C DATA **
 90  DATA 166,251      : REM $A6 $FB        -LDX $FB
100  DATA 189,0,96     : REM $BD, $00, $60  -LDA $6000,X
110  DATA 133,252      : REM $85, $FC       -STA $FC
120  DATA 96           : REM $60            -RTS
130  :
140  REM ** CALCULATE TABLE VALUES **
150  FOR C = 0 TO 100
160  F = (1.8 * C) + 32
170  POKE 24576 + C, F
180  NEXT C
190  :
200  HOME
210  INPUT "CENTIGRADE VALUE"; C
220  POKE 251, C
230  CALL CODE
240  PRINT "FAHRENHEIT VALUE";
250  PRINT PEEK (252)
```

Lines 150 to 180 calculate the equivalent Fahrenheit values for Centigrade values in the range 0-100. Each value is POKEd in turn into memory, to form a table which has its base at 24576 ($6000). The input Centigrade value for conversion is POKEd into location 251 ($FB), and subsequently loaded into the X register when the machine code is executed (line 90). This value is used as the index to the table when loading the accumulator with the corresponding Fahrenheit value (line 100). This Fahrenheit value is stored in location 252 ($FC) so that it can be accessed by the calling BASIC. The following example illustrates this graphically.

Example: Convert 3°C to Fahrenheit.





Therefore 3°C is equivalent to 37°F.

18 Floating a Point

So far throughout this introduction to assembly language programming we have only been concerned with *integers*, or whole numbers. As in the real world though, *floating point* numbers also exist in the machine code world. A floating point number is one that contains a decimal point (although in binary this is more correctly referred to as a 'bicimal' point).

For example, the denary number 5.25 is a floating point number whereas the number 7 is a whole or integer number. In Applesoft BASIC the binary floating point numbers have what is known as 10 digit precision, displayed with 9 digits and with exponents in the range +38 to -39. The exponent of a number is simply a scientific notational form of representing numbers. For example, the number 1234.567 could be expressed exponentially as:

0. 1234567E+4

The 'E' denotes the exponential value and the +4 the fact that the decimal point has been moved four places in a positive direction. Another way of writing this exponentially is:

0.1234567×10^4

Similarly, the decimal value 0.0000123 can be expressed as 0.123E-4 or 0.123×10^{-4} the -4 indicating that the decimal point has been moved four places in a negative direction.

THE FLOATING POINT ACCUMULATORS

The 6502 is provided with two memory-mapped floating point accumulators which manipulate the floating point numbers. These are known as the FAC (Floating Point Accumulator) and the ARG (ARGument register) also known as FAC#1 and FAC#2. The addresses associated with them in zero page are shown in Table 18.1.

Looking at the two floating point accumulators we can see that each has six associated bytes. As already mentioned, each value has 10 digit precision, so to enable the value to be packed into six bytes it must be broken down into two components called the *binary mantissa* and the *binary exponent*. These are illustrated in

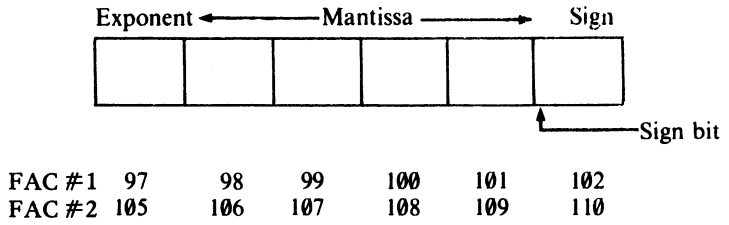


Figure 18.1 Floating point accumulator architecture.

Table 18.1

Label	Address		Description
	Decimal	Hex	
FACEXP	157	$009D	FAC#1 exponent
FACHO	158-161	$009E-$00A1	FAC#1 mantissa
FACSGN	162	$00A2	FAC#1 sign
ARGEXP	165	$00A5	FAC#2 exponent
ARGHO	166-169	$00A6-$00A9	FAC#2 mantissa
ARGSGN	174	$00AE	FAC#2 sign

The binary mantissa is the 'number' part of the value, and this is stored in the center four bytes of the FAC (and ARG). The sign of the number denotes whether it is a positive or negative value, and this is stored in the sixth byte of the FAC (or ARG). Only a single bit (bit 7) is required to store the sign—'1' represents a negative mantissa and '0' a positive mantissa.

The binary exponent is the first byte of the FAC (and ARG) and this is used to represent both positive and negative exponent values by adding the value to, or subtracting the value from, 128. For example, an exponent of +15 is represented by:

$+15 = 128 + 15 = 143$

Whereas a negative exponent of -15 is expressed as:

$-15 = 128 - 15 = 113$

To allow a variety of floating point numbers to be handled by a standard set of floating point subroutines, the Apple //'s Applesoft BASIC interpreter normalizes them to a representation such that the most significant bit is always a '1'. Consider the hexadecimal number $0345. Writing this in binary form we obtain:

$0345 = 0000 0011 0100 0101

In this case, the binary now has an exponent value of 2, or more correctly 1^2 with the bicimal point being at the far right of the number. If we were to express this properly in exponential form we would read:

$0000 0011 0100 0101 \times 2^0$

Now, to normalize this value we need to 'float' the bicimal point along to the left until it sits in front of the leftmost '1'. Figure 18.2 shows the process. Now, if we count the number of shifts we obtain our exponent value, which in this case is 10. Thus:

$0345 =0.1101 0001 0100 0000 X 2^{10}

To represent this in either of the floating point accumulators we must add the exponential value to 128 giving an exponent value of:

128 + 10 = 138 = $8A (1000 1010)

Figure 18.2 floating the bicimal point.

Moving back to the mantissa, we have four bytes to fill, therefore the least significant bits must be padded out with 0s to give:

1101 0001 0100 0000 0000 0000 0000 0000

Finally, to complete our normalization of $0345 we must indicate a positive value by placing a 0 in bit 7 of the sign byte. Our final representation of $0345 is given by:

Exponent	1st byte	1000	1010	$8A	
Mantissa	2nd byte	1101	0001	$D1	
	3rd byte	0100	0000	$40	
	4th byte	0000	0000	$00	(padded bytes)
	5th byte	0000	0000	$00	
Sign	6th byte	0xxx	xxxx		

The xs in the sign byte denote that these bits may have any value.

The example given above was really an integer one, but numbers that do contain bicimal values can be normalized in exactly the same manner. For instance, the decimal value 255.75 can be expressed in binary terms as:

1111 1111 . 11

This can be normalized as before, giving a binary exponential representation of:

.1111 1111 1100 0000 X 2^{8}

USING USR

A further Applesoft BASIC statement, USR, is provided to call sections of machine code. It has an advantage over a normal CALL in that it can also pass data from

Applesoft BASIC into the floating point accumulator, so that it can be manipulated by a user-supplied machine code program before returning the result to the calling Applesoft program. Before executing USR the address of a machine code subroutine must be seeded into USRADD, which is located at the two bytes starting at 11 ($000B). A USR call can take two forms:

```
USR (NUM)
```

transfers the value assigned to the variable NUM (or any other specified variable) into FAC#1 before handing control to the machine code routine located at the vectored address in USRADD. Whereas:

```
A = USR (B)
```

places the contents of B into FAC#1, executes the machine code at USRADD and returns the final result, via FAC#1, in the variable B.

Let's have a look at a couple of simple examples to get things clear in our minds.

Program 37

```
 10  REM ** USR DEMO **
 20  REM ** SET UP DUMMY MACHINE CODE **
 30  POKE 768,96 : REM RTS OPCODE
 40  HOME
 50  POKE 11,0 : REM SET UP USRADD TO
 60  POKE 12,3 : REM POINT TO 768
 70  A = 0 : B = 837
 80  PRINT "PRE USR A = ";A
 90  A = USR (B)
100  PRINT "POST USR A = ";A
```

This program begins by POKEing an RTS instruction into memory at location 768 (line 30). USRADD is then pointed to this location. The values of A and B are assigned (line 70) and the value of the former printed (line 80). The USR routine is then called (line 90) passing the value of B into FAC#1. The code pointed to by USRADD is then executed—it returns control immediately back to Applesoft BASIC (remember it's just RTS) passing the value in FAC#1 into the variable A which is subsequently printed out (line 100).

We can modify this program slightly so that it actually does something! Add the following two lines:

```
25  POKE 768,230
26  POKE 769,158    :  REM INC $9E
```

now change line 30 to read:

```
30  POKE 770,96     :  REM RTS
```

RUN the program. The printed result should be:

```
POST USR A = 841
```

which is 4 more than the value passed into it through B (which was 837). What happened here was that the two new bytes of machine code (lines 25 and 26) incremented the high byte of the FAC# 1 mantissa located at location 158. But why should this add 4 rather than 1? Examining the binary will make things clearer. 837 is $0345 which is the value we were working on earlier. We know from our previous calculations that the byte stored in location 158 was $D1. Incrementing this gives $D2, therefore the four bytes of the FAC#1 mantissa read:

$D2	$40	$00	$00
1101 0010	0100 0000	0000 0000	0000 0000

We also know from our earlier calculations that the exponential value was 2^{10}. Floating the bicimal point ten places to the right to return to an un-normalized position gives:

1101001001.00000000000000000000

Ignoring the non-significant zeros and sorting the binary into bytes we obtain:

0000 0011	0100 1001
$03	049

and of course $0349 = 841.

The important point to remember when dealing directly with the FAC is that we are handling normalized values and even something as seemingly simple as an increment instruction will not have the obvious result! Another important point to remember is that the contents of the FACs cannot be examined directly from Applesoft BASIC by PEEKing locations. This is because even an operation as straightforward as PEEK will affect the FAC's contents. Therefore, to look at the contents of a FAC you need a machine code routine such as Program 38.

Program 38

```
 10  REM ** SAVE FAC#1 **
 20  CODE = 768
 30  FOR LOOP = 0 TO 10
 40      READ BYTE
 50      POKE LOOP + CODE, BYTE
 60  NEXT LOOP
 70  :
 80  REM ** M/C DATA **
 90  DATA 162,6        : REM $A2, $06        -LDX #6
100                     REM AGAIN
110  DATA 181,156      : REM $B5, $9C        -LDA $9C,X
120  DATA 157,52,3     : REM $9D, $34, $03   -STA $0334,X
130  DATA 202          : REM $CA             -DEX
140  DATA 208,248      : REM $D0, $F8        -BNE AGAIN
150  DATA 96           : REM $60             -RTS
160  :
170  HOME
180  POKE 11,0         : REM SET USRADD POINTING
190  POKE 12,3         : REM TO 768 ($0300)
```

```
200   B = 837              :  REM VALUE TO PASS TO FAC#1
210   A = USR (B)          :  REM PASS AND EXECUTE CODE
220   PRINT "A = ";A
230   PRINT "FAC#1 = ";
240   FOR X = 1 TO 6
250   PRINT PEEK (820 + X);" ";
260   NEXT X
```

The machine code to save the contents of FAC#1 is quite simple, and just involves an indexing loop which first loads a byte into the accumulator and then stores it somewhere safe (lines 90 to 150). RUNNING the program produces the following output:

```
A = 837
FAC#1 = 138 209 64 0 0 81
```

The first five bytes compare favorably with the calculated values above. The final byte is derived from FACSGN. The value 81 is $51 and in binary is 0101 0001. The sign bit, bit 7, is clear and denotes a positive number. We can change the sign to a negative value by 'forcing' bit 7 to 1. To do this we need to logically OR the contents of FACSGN with $80, 10000000 binary. Add the following lines to the program:

```
82   DATA 165,162     :  REM $A5, $A2     —LDA FACSGN
83   DATA 9,128       :  REM $09, $80     —ORA #$80
84   DATA 133,162     :  REM $85, $A2     —STA FACSGN
```

and change the loop count to:

```
30   FOR LOOP = 0 TO 16
```

Now RUN the program. The result returned in A is now -837, while FACSGN returns 209.

INTEGER TO FLOATING POINT

Included in the built-in subroutines are several which allow numbers to be converted from integer to floating point and vice versa. These can be of great help in allowing us to manipulate multibyte numbers in machine code, so let's examine a few of them in operation.

Program 39 shows how an integer value can be converted into its normalized floating point counterpart. The subroutine to do this is located at 58098 ($E2F2). The integer value is expected to be found in the accumulator (high byte) and Y register (low byte) — on completion of the subroutine the floating point value can be extracted from FAC#1.

Program 39

```
10   REM ** INTEGER TO FP **
20   CODE = 768
```

```
 30   FOR LOOP = 0 TO 17
 40       READ BYTE
 50       POKE LOOP + CODE, BYTE
 60   NEXT LOOP
 70   :
 80   REM ** M/C DATA **
 90   DATA 169,1        : REM $A9, $01        -LDA #1
100   DATA 160,35       : REM $A0,$23         -LDY #$23
110   DATA 32,242,226   : REM $20, $F2, $E2   -JSR $E2F2
120   DATA 162,6        : REM $A2, $06        -LDX #6
125                       REM AGAIN
130   DATA 181,156      : REM $B5, $9C        -LDA $9C,X
140   DATA 157,52,3     : REM $9D, $34, $03   -STA $0334,X
150   DATA 202          : REM $CA             -DEX
160   DATA 208,248      : REM $D0, $F8        -BNE AGAIN
170   DATA 96           : REM $60             -RTS
180   :
190   HOME
200   CALL CODE
220   PRINT "FAC#1 = ";
230   FOR X = 1 TO 6
240       PRINT PEEK (820 + X);" ";
250   NEXT X
```

The integer value being converted here is $0123, and the high and low bytes are placed in the appropriate registers (lines 90 and 100) before the conversion routine is called (line 110). The floating point value is extracted from FAC#1 (lines 120 to 160) so that it can be PEEKed by the BASIC loop. RUNning the program produces this result:

```
FAC#1 = 137 145 128 0 0 0
```

Evaluating this, we have an exponent of 9 (137 - 128), and two bytes which in binary form give:

```
1001 0001    1000 0000
  145          128
```

Moving the bicimal point 9 places to the right we arrive at a final value of:

```
0000 0001 0010 0011
  $01        $23
```

$0123, which was our original value. The conversion works!

This subroutine for integer to floating point conversion can only handle numbers in the range 0 to 32767 ($7FFF). This is because bit 15 is used to determine the sign of the integer value to be converted. If it is clear, then a positive value is assumed; if it is set, a negative value is evaluated and the fact signaled in FACSGN. To see this, try changing the following lines:

```
 90   DATA 169,255      : REM $A9, $FF        -LDA #$FF
100   DATA 160,255      : REM $A0, $FF        -LDY #$FF
```

RUN the program now and see what happens—I'll leave it up to you to work out how and why you got the result you did!

FLOATING POINT TO INTEGER

A subroutine which operates in the reverse direction, and converts a floating point value in FAC#1 to an integer and stores it in 80 and 81 ($50 and $51), is located at 59218 ($E752).

Program 40

```
 10  REM ** FP TO INTEGER **
 20  CODE = 768
 30  FOR LOOP = 0 TO 17
 40      READ BYTE
 50      POKE LOOP + CODE, BYTE
 60  NEXT LOOP
 70  :
 80  REM ** M/C DATA **
 90  DATA 162,6        : REM $A2, $06       -LDX #6
100                     REM AGAIN
110  DATA 189,52,3     : REM $BD, $34, $03  -LDA 820,X
120  DATA 149,156      : REM $95, $9C       -STA $9C,X
130  DATA 202          : REM $CA            -DEX
140  DATA 208,248      : REM $D0, $F8       -BNE AGAIN
150  DATA 32,82,231    : REM $20, $52, $E7  -JSR $E752
160  DATA 133, 251     : REM $85, $FB       -STA $FB
170  DATA 132,252      : REM $84, $FC       -STY $FC
180  DATA 96           : REM $60            -RTS
190  :
200  FOR X = 0 TO 5
210      READ FAC
220      POKE 821 + X, FAC
230  NEXT X
240  REM ** FAC DATA **
250  DATA 137, 145, 128, 0, 0, 0
260  HOME
270  CALL CODE
280  PRINT "RESULTS RETURNED ARE: "
290  PRINT "ACCUMULATOR = "; PEEK (251)
300  PRINT "Y REGISTER = "; PEEK (252)
```

This program passes the normalized value of $0123 (line 250) into FAC#1. The DATA is READ by the loop in lines 210 to 230 and placed into unused memory from 820. The machine code begins by transferring this data into FAC# 1 using indexed addressing (lines 90 to 140). Once there, the conversion routine is called (line 150) and the resultant values in the A and Y registers saved in zero page, from whence they can be PEEKed (lines 290 and 300). RUNning the program produces:

```
RESULTS RETURNED ARE:
ACCUMULATOR = 1
Y REGISTER = 35
```

Converting this to hex gives $0123!

FLOATING MEMORY

Several subroutines are available which allow floating point values to be transferred to and fro between either of the floating point accumulators and memory. However, before we can use these, we must see how floating point numbers are stored in memory, as a slightly different format is used. Let's go back to the normalized value of $0345 we used earlier, this was stored in the FAC as:

Exponent	$8A	1000 1010
Mantissa	$D1	1101 0001
	$40	0100 0000
	$00	0000 0000
	$00	0000 0000
Sign		0xxx xxxx

Looking at this, we can see that we waste 7 bits of the final sign byte simply because we only use bit 7 to denote the sign. If another way could be found of encoding this then the sign byte could be dispensed with.

You will remember that to normalize a floating point value the bicimal point is floated left continuously until it reaches the left-most 1. We know, therefore, that the first bit of the mantissa will always be a 1. As the Applesoft BASIC Interpreter knows this too, it can 'forget' the 1 and use this bit to store the sign. When the interpreter converts a floating point number stored in memory (outside either floating point accumulator), it looks at the msb of the mantissa to evaluate the sign, then resets it back to 1 to evaluate the number proper! In memory, then, the normalized representation of $0345 is compacted into just 5 bytes stored thus:

Exponent	$8A	1000 1010	
Mantissa	$51	0101 0001	(bit 7 = 0 therefore number positive)
	$40	0100 0000	
	$00	0000 0000	
	$00	0000 0000	

19 Speeding Up and Slowing Down

At the beginning of this book I said that one of the advantages of using machine code was that it runs very much faster than an interpreted language such as BASIC. But just how fast is fast, and can we calculate the amount of time a piece of machine code takes to execute? The answer is yes, and we shall now see just how to do it.

The Apple // has within its plastic case its own clock which it uses to 'tick' out the stages of an instruction's execution. This clock takes the form of a quartz crystal oscillator which produces a signal whose frequency is divided down to 1 MHz, which simply means it 'ticks' or cycles 1 million times every second. Appendix 4 lists the number of cycles each instruction requires. As can be clearly seen, the more complex forms of addressing modes take longer to execute. The fastest instructions, generally employing implied or immediate addressing, take just 2 cycles to complete. In real terms this is 0.000002 seconds or more simply, 2 microseconds—two millionths of a second! Instructions which access absolute addresses can vary in execution time. For example, instructions using absolute indexed addressing will generally take 4 cycles to complete, but, if a memory page boundary is crossed an extra cycle is required in order to increment the high byte of the Program Counter (PCH).

Branch instructions can take 2, 3 or 4 cycles. If the branch does not take place, then only two cycles are needed. Three cycles are needed if the branch occurs, and a further cycle if a page boundary is crossed.

Another interesting point to note is that while PHA and STA ZEROPAGE both take 3 cycles, PLA takes 4 cycles compared with LDA ZEROPAGE which only takes 3 cycles. It is therefore quicker to use zero page for storage rather than the stack. It should be remembered, though, that PHA : PLA require only two bytes whereas STA ZEROPAGE: LDA ZEROPAGE require four—so there has to be a trade-off in memory requirements against execution time.

Using Appendix 4 we can calculate some program execution times.

```
LDX #$FF        \ immediate
TXA             \ implied
AND $FB         \ zero page
STA $1500, X    \ absolute indexed
JMP ($FC)       \ indirect
```

The first instruction, LDA #$FF, uses immediate addressing and therefore requires only 2 cycles. Similarly the implied TXA instruction needs only 2 cycles. AND $FB takes a little longer, 3 cycles, as it has to access the contents of a zero page location.

The absolute indexed addressing of STA $1500, X can take either 4 or 5 cycles, depending on whether a page boundary is crossed. The shorter time is the correct one here because the base address ($1500) is itself on a page boundary, which means the index register cannot contain a value great enough to cross the next page boundary. Finally, the indirect jump takes 3 cycles. The total time of program operation is therefore 2 + 2 + 3 + 4 + 3 = 14 cycles.

Consider the following simple loop.

```
        LDY #$FF        \ 2 cycles
LOOP    DEY             \ 2 cycles
        BNE LOOP        \ 3 cycles
```

The Y register loading instruction takes 2 cycles and the loop execution a total of 5 cycles. However, the LOOP will execute 255 times and we must of course take this into account. This gives a total LOOP execution period of 5 x 255 - 1 = 1274 cycles. The 'minus one' in the calculation is because the 'last' branch will not occur and will therefore only require 2 cycles of computer time. If the branch causes a page boundary crossing, then each execution of LOOP will need 6 cycles, giving a total of 6 x 255 - 2 = 1528 cycles for the program (the 'minus two', of course, is for the last 'page boundary branch' which does not take place).

Because of the speed of machine code it is sometimes necessary to produce deliberate delays to slow things down to allow mere humans time to interact with the Apple //. There is an instruction that does nothing but provide a 2 cycle delay:

```
NOP     No operation
```

To demonstrate its use let's write a piece of machine code to produce about a two millisecond delay (0.002 seconds or 2000 cycles). The process has to be worked out by trial and error. Using the loop described above and inserting a single NOP command would give:

```
        LDY #$FF        \ 2 cycles
LOOP    NOP             \ 2 cycles
        DEY             \ 2 cycles
        BNE LOOP        \ 3 cycles
```

Total loop time is 7 cycles, which gives a total execution time of 7 x 255 - 1 = 1784 cycles. Because LDY #$FF is fundamental to the LOOP operation we should include this—giving a total delay of 1786 cycles. This falls short of the desired 2000 cycles.

By adding an extra NOP, the delay is increased to 9 x 255 - 1 + 2 = 2296 cycles. This is too high but can be reduced by altering the value of the loop counter.

```
        LDY #$DE        \ 2 cycles
LOOP    NOP             \ 2 cycles
        NOP             \ 2 cycles
        DEY             \ 2 cycles
        BNE LOOP        \ 3 cycles
```

This final version produces a delay which is just one half of one millionth of a second short of a millisecond i.e. 9 x 222 - 1 + 2 = 1999 cycles. You might like to try experimenting with loops to produce a delay of exactly one millisecond.

20 Interrupts and Breaks

INTERRUPTS

An interrupt is a signal that causes the program that is currently running to halt temporarily while program control is transferred to a subroutine somewhere in memory that is designed to service the interrupt. Once the interrupt has been dealt with control is passed back to the original program, allowing it to continue as though nothing had happened.

There are two different types of interrupt — the NMI (non-maskable interrupt) and the IRQ (interrupt request). The difference between the two is that an NMI must be serviced immediately because it is too important to ignore, whereas an IRQ can be ignored until we are ready to service it. A variety of different devices can interrupt the 6502 .

On the Apple //, the NMI is almost never used. As this type of interrupt cannot be 'programmed' directly it is not specifically covered here—though much of what follows is applicable. The IRQ has two instructions associated with it which directly affect bit 2 of the Status register, they are:

CLI	Clear interrupt disable bit
SEI	Set interrupt disable bit

The condition of the Interrupt flag within the Status register determines whether the IRQ is serviced or ignored when it occurs. If the Interrupt bit is set (I=1) then an IRQ is ignored; if the Interrupt bit is clear (I=0) an IRQ is serviced the moment it occurs.

Let's examine the exact sequence of events that takes place when an IRQ occurs, assuming that the Interrupt flag is clear. Firstly, the processor completes the operation specified by the machine level instruction it is currently executing. The Status register is examined to determine if bit 2 is clear (in our case it is), in which case the contents of the Program Counter and Status register are pushed on to the hardware stack. The Interrupt bit is now set (I= 1) to shut out any further IRQs while one is being serviced. Note that the Interrupt bit is set after the Status register has been saved, thus preserving its pre-interrupt condition. At the end of this chain of events, the Program Counter is loaded with the contents of the locations **$FFFE** and **$FFFF**, the top two bytes in the memory map, and a jump performed to this address. The machine code located here (64134) is listed below, and ends with a jump to the actual Interrupt service routine responsible for locating and servicing the IRQ.

The IRQ routine also has a vector in block zero RAM associated with it—at 1022 ($03FE)—and any user-generated interrupt routines should gain control through here. On completion of the interrupt service routine, control must be returned to the interrupted program. To facilitate this a further instruction is provided:

RTI Return from interrupt

This instruction resets the Status register and Program Counter to the values previously saved on the stack, and allows the original program to continue from the point at which it was interrupted.

Before performing the indirect jump to the IRQ vector at $3FE, the monitor ROM checks to see if the call to the IRQ handler was caused by a BRK instruction. If it was, the contents of all of the registers is automatically saved. If it wasn't, however, the contents are not saved. Your routine that is called when an interrupt occurs must do this first before it does anything else. This is important because they will undoubtedly be altered by the interrupt service routine, and we need to ensure that the contents of all registers are in their pre-interrupt condition before the RTI is executed. The machine code located at 64134 responsible for this reads as follows:

```
64134   STA ACC          — save the accumulator
64136   PLA              — load accumulator from stack
64137   PHA              — restore stack
64138   ASL A
64139   ASL A
64140   ASL A
64141   BMI BREAK        — test for BREAK ($FA92)
64143   JMP ($3FE)       — jump to IRQ service
```

BREAKS

There is an instruction in the Apple //'s 6502 instruction set which allows a software type of interrupt to be generated—this instruction is:

BRK Break

BRK is a single byte instruction (opcode $00) which can be inserted into programs as and when required. When the 6502 encounters a BRK instruction it does a number of things—in fact, it proceeds along a similar path to that taken by an IRQ. Firstly it increments the Program Counter so that it is now pointing to the instruction after the BRK, and then pushes this two byte address on to the hardware stack. Next it sets the Break flag, which is bit 4 of the Status register, and then pushes this on to the stack before jumping to the BRK servicing routine. This routine's address is stored in locations $FFFE and $FFFF and therefore it is the same servicing routine as that used by an IRQ. Or is it? Well not exactly—it just enters at the same point.

After storing what's in the accumulator, this routine retrieves the last value (the Status register) that was pushed onto the stack to see if a BRK instruction was encountered. If it was, the N flag will be set. The three ASL instructions are used to properly locate this bit for the BMI test. If it was set, the routine passes control to the BRK handler which is located at $FA92.

The BRK handler allows you to perform simple machine code debugging by saving all the registers and then printing out their contents at the time the BRK occurred, on the screen.

21 Prepacked Utilities

I am sure that you will find the programs that follow in this chapter very useful when you write your own serious machine code—thus I have called them utility programs because they have a practical use. Included are programs to:

1. Convert an ASCII based hex number into its binary equivalent.
2. Convert and print a binary value as a two digit ASCII based hex number.
3. Print an ASCII string stored within the machine code itself.

HEX TO BINARY CONVERSION

The following routine will convert two ASCII based hexadecimal characters into their eight bit binary equivalent. For example, if the characters F E are input, the binary value returned would be 1111 1110—this will of course be printed as 254, the decimal equivalent. This is a particularly important procedure especially if programs handling hexadecimal data are anticipated. Conversion is not difficult and Table 21.1 gives some indication of what is required.

Table 21.1

Hex character	Binary value	ASCII value	ASCII binary
0	0000	$30	0011 0000
1	0001	$31	0011 0001
2	0010	$32	0011 0010
3	0011	$33	0011 0011
4	0100	$34	0011 0100
5	0101	$35	0011 0101
6	0110	$36	0011 0110
7	0111	$37	0011 0111
8	1000	$38	0011 1000
9	1001	$39	0011 1001
A	1010	$41	0100 0001
B	1011	$42	0100 0010
C	1100	$43	0100 0011
D	1101	$44	0100 0100
E	1110	$45	0100 0101
F	1111	$46	0100 0110

In the case of the characters 0 to 9 it should be fairly obvious that all we need to do to convert them to binary is to mask out the high nibble of the character's ASCII code, because the low nibble binary is the same as the hex character itself.

Converting the characters A to F is a little less obvious. However, if the high nibble of the ASCII code is masked off, then the remaining low nibble is 9 less than the hex value required. For example, the ASCII for 'D' is 01000100, masking off the high nibble gives 0100, which is 4, add 9 to this to arrive at $D = 1101.

Program 41

```
 10  REM ** ASCII HEX TO BINARY **
 20  CODE = 768
 30  FOR LOOP = 0 TO 54
 40      READ BYTE
 50      POKE CODE + LOOP, BYTE
 60  NEXT LOOP
 70  :
 80  REM ** M/C DATA **
 90  DATA 32,34,3      : REM $20, $22, $03    -JSR CHARACTER
100  DATA 165,252      : REM $A5, $FC         -LDA $FC
110  DATA 32,24,3      : REM $20, $18, $03    -JSR CHECK
120  DATA 10           : REM $0A              -ASL A
130  DATA 10           : REM $0A              -ASL A
140  DATA 10           : REM $0A              -ASL A
150  DATA 10           : REM $0A              -ASL A
160  DATA 133,253      : REM $85, $FD         -STA $FD
170  DATA 165,251      : REM $A5, $FB         -LDA $FB
180  DATA 32,24,3      : REM $20, $18, $03    -JSR CHECK
190  DATA 5,253        : REM $05, $FD         -ORA $FD
200  DATA 133,254      : REM $85, $FE         -STA $FD
210  DATA 96           : REM $60              -RTS
220  REM ** CHECK SUBROUTINE : $0318 **
230  DATA 201,186      : REM $C9, $3A         -CMP #$BA
240  DATA 176,3        : REM $B0, $03         -BCS ATOF
250  DATA 41,15        : REM $29, $0F         -AND #$0F
260  DATA 96           : REM $60              -RTS
270                      REM ATOF
280  DATA 233,55       : REM $E9, $37         -SBC #$37
290  DATA 96           : REM              $60-RTS
300  REM ** CHARACTER SUBROUTINE : $0322 **
310                      REM FIRST
315  DATA 169,32       : REM $A9, $20         -LDA #$20
320  DATA 32,27,253    : REM $20, $1B, $FD    -JSR $FD1B
330  DATA 32,237,253   : REM $20, 237, 253    -JSR $FDED
340  DATA 133,252      : REM $85, $FC         -STA $FC
350                      REM SECOND
355  DATA 169,32       : REM $A9, $20         -LDA #$20
360  DATA 32,27,253    : REM $20, $1B, $FD    -JSR $FD1B
370  DATA 32,237,253   : REM $20, $ED, $FD    -JSR $FDED
380  DATA 133,251      : REM $85, $FB         -STA $FB
390  DATA 96           : REM $60              -RTS
400  :
410  HOME
420  PRINT "ENTER TWO HEX DIGITS: ";
430  CALL CODE
435  PRINT
```

```
440   PRINT "THEIR BINARY VALUE IS: ";
450   PRINT PEEK (254)
```

The program begins by calling the CHARACTER subroutine to obtain two ASCII based hex characters (lines 310 to 390) and places them in locations 251 ($FB) and 252 ($FC). The high nibble character is converted first by calling the CHECK subroutine (lines 230 to 290). If the ASCII based character byte is in the range 0-9 the high nibble is masked off (line 250), otherwise 55 is subtracted from it (line 280). This has the same effect as masking off the high nibble and adding nine! On returning from the CHECK subroutine the result, now held in the low four bits, is shifted left into the high nibble of the accumulator (lines 120 to 150) and saved for future reference (line 160).

A similar procedure is used to convert the low ASCII based character, but on return from the CHECK subroutine the resultant binary is logically ORed with the previous result (line 190) to produce the final value. This is then stored in location 254 ($FD).

This program could be improved in a number of ways; for instance, the ASCII characters are not echoed to the screen, nor are there any checks to ensure that only legal hex values are entered. You might like to add these extra facilities yourself?

BINARY TO HEX CONVERSION

To convert an eight bit binary number into its ASCII hex equivalent characters, the process described above is reversed. However, because characters are printed on the screen from left to right we must, in this instance, deal with the high nibble of the byte first. Program 42 requests a number for conversion (lines 300-370) and holds it in the accumulator. It is then pushed onto the stack (line 90), and the high nibble is shifted four times to move it into the low nibble position (lines 100-130). The subroutine FIRST does the conversion. After ensuring that no high bits are set (line 170) the binary value is tested to see if it's in the range 0-9 (line 180). If itis not (and is therefore in the range A-F), 7 is added to the accumulator (line 200—6 by the command plus 1 from the Carry flag). Line 220 performs the conversion by adding $30, which effectively sets bits 4 and 5. After printing the ASCII character (line 230) control returns back to line 150, where the original binary value is pulled off the stack in readiness for the low nibble (line 170) to be converted into the appropriate ASCII character.

Program 42

```
 10   REM ** PRINT ACCUMULATOR AS HEX NUMBER **
 20   CODE = 768
 30   FOR LOOP = 0 TO 21
 40        READ BYTE
 50        POKE CODE + LOOP, BYTE
 60   NEXT LOOP
 70   :
 80   REM ** M/C DATA **
 90   DATA 72          : REM $48          —PHA
100   DATA 74          : REM $4A          —LSR A
110   DATA 74          : REM $4A          —LSR A
120   DATA 74          : REM $4A          —LSR A
130   DATA 74          : REM $4A          —LSR A
140   DATA 32,9,3      : REM $20, $09, $03   —JSR FIRST
```

```
150  DATA 104          : REM $68              -PLA
160  REM ** FIRST SUBROUTINE : $0309 **
170  DATA 41,15        : REM $29, $0F         -AND #$0F
180  DATA 201,10       : REM $C9, $0A         -CMP #$0A
190  DATA 144,2        : REM $90, $02         -BCC OVER
200  DATA 105,6        : REM $69, $06         -ADC #$06
210                      REM OVER
220  DATA 105,176      : REM $69, $B0         -ADC #$B0
230  DATA 76,237,253   : REM $4C, $ED, $FD    -JMP $FDED
240  :
250  REM ** DEMO PROGRAM **
260  REM LDA $FB : JMP $0300
270  POKE 828, 165 : POKE 829, 251
280  POKE 830, 76 : POKE 831, 0 : POKE 832, 3
290  HOME
300  PRINT "HIT A KEY AND ITS HEX VALUE IN "
310  PRINT "ASCII WILL BE DISPLAYED"
320  GET A$
330  IF A$=" " THEN GOTO 320
340  A = ASC(A$)
350  POKE 251, A
360  REM ** CALL LINK ROUTINE-LINES 270 AND 280 **
370  CALL 828
380  REM ** CALL 'CALL CODE' TO USE DIRECTLY **
```

Program 42 is demonstrated by pressing any of the alphanumeric keys—it then prints their ASCII hexadecimal value.

OUTPUT ASCII STRING

This utility subroutine—Program 43—enables ASCII character strings to be stored within the body of machine code programs ready for printing on to the screen. It has two advantages over the normal absolute indexing approach. Firstly, it is inserted into the program at the point it is needed and secondly, it calculates its own address and is therefore fully relocatable.

Program 43

```
10   REM ** ASCII STRING OUTPUT ROUTINE **
20   CODE = 768
30   FOR LOOP = 0 TO 28
40       READ BYTE
50       POKE CODE + LOOP, BYTE
60   NEXT LOOP
70   :
80   REM ** M/C DATA **
90   DATA 104          : REM $68              -PLA
100  DATA 133,251      : REM $85, $FB         -STA $FB
120  DATA 104          : REM $68              -PLA
130  DATA 133,252      : REM $85, $FC         -STA $FC
140                      REM REPEAT
150  DATA 160,0        : REM $A0, $00         -LDY #$00
160  DATA 230,251      : REM $E6, $FB         -INC $FB
```

```
170   DATA 208,2          :  REM $D0, $02          -BNE CLEAR
180   DATA 230,252        :  REM $E6, $FC          -INC $FC
190                          REM CLEAR
200   DATA 177,251        :  REM $B1, $FB          -LDA ($FB), Y
210   DATA 48,8           :  REM $30, $08          -BMI FINISH
215   DATA 73,128         :  REM $49, $80          -EOR #$80
220   DATA 32,237,253     :  REM $20, $ED, $FD     -JSR $FDED
230   DATA 76,6,3         :  REM $4C, $06, $03     -JMP REPEAT
240                          REM FINISH
250   DATA 108,251,0      :  REM $6C, $FB, $00     -JMP ($00FB)
260   :
270   REM ** DEMO ROUTINE **
280   REM ** LOCATED AT $031D **
290   DEMO = 797
300   FOR LOOP = 0 TO 1
310       READ BYTE
320       POKE DEMO + LOOP, BYTE
330   NEXT LOOP
340   :
350   REM
360   DATA 32,88,252      :  REM $20, $58, $FC     -JSR $FC58
370   DATA 32,0,3         :  REM $20, $00, $03     -JSR OUTPUT
380   REM ** NOW STORE ASCII CODES FOR PRINTING **
390   DATA 65,80,80,76,69,32,47,47,13
400   REM      A, P, P, L, E, , /, /,<CR>
410   DATA 234            :  REM $EA               -NOP
420   DATA 96             :  REM $60               -RTS
430   CALL DEMO
```

The main ASCII output routine is between lines 90 and 250, a short demonstration program is included in lines 360 to 420. The demo program begins by clearing the screen (line 360), then the OUTPUT routine located at $0300 is called. Immediately following this call the ASCII text for output (which in this case is the phrase Apple //) is POKEd into memory. The end of the string is marked by a negative byte—one that has its most significant bit set. NOP is ideal for this because it doesn't do anything (line 410)! The ASCII print routine, which is just 29 bytes long, begins by pulling the RTS address (from the calling subroutine) off the stack and placing it into two zero page locations, 251 ($FB) and 252 ($FC).

Because the string immediately follows the CODE subroutine call (see Figure 21.1), post-indexed indirect addressing can be used to load the first string character into the accumulator (line 200). Line 210 tests to see if the string-terminating negative byte has been reached. If not, the character is exclusive-or'ed with the hex number $80 so that it will appear as normal white-on-black text (line 215) and then it is printed (line 220). A JMP back to REPEAT is implemented (line 230) and the zero page address incremented (lines 160-180) so that the next string character can be sought out. Once the negative byte is encountered and the test of line 210 succeeds, an indirect jump (line 250) via the zero page address will return control to the calling machine code program.

	Address	Hex	Mnemonic/character
797	$031D	20	JSR
798	$031E	58	$FC58
799	$031F	FC	
800	$0320	20	JSR
801	$0321	00	$0300
802	$0322	03	
803	$0323	41	A
804	$0324	50	P
805	$0325	50	P
806	$0326	4C	L
807	$0327	45	E
808	$0328	20	
809	$0329	2F	/
810	$032A	2F	/
811	$032B	0D	<CR>
812	$033C	EA	NOP
813	$033D	60	RTS

Figure 21.1 Memory layout of part of Program 43.

Appendices

1 The Screen

The character set can be displayed on the screen in two different ways:

1. By printing the ASCII code.
2. By storing the screen code for the character into screen memory.

The screen code should be either 64 or 128 greater than the ASCII equivalent. Thus if you wanted to print an A, whose ASCII code is 65, you'd store either 129 or 193 to the location you want it to appear in on the screen.

To print an ASCII code on to the screen, first load the ASCII code into the accumulator and then call the monitor COUT routine at $FDED. For example to print an 'A' use:

```
LDA #65          — ASCII code for A
JSR $FDED        — print it
```

The print position can be specified by first loading the cursor's vertical and horizontal position into CV ($25) and CH ($24).

Using screen codes is almost as easy. All you have to do is load the accumulator with the desired screen code and then store it in the appropriate memory location. For example, to display an 'A' at the first position of the first screen line, the following can be used:

```
LDA #129         — POKE code for A
STA #1024        — store in screen memory
```

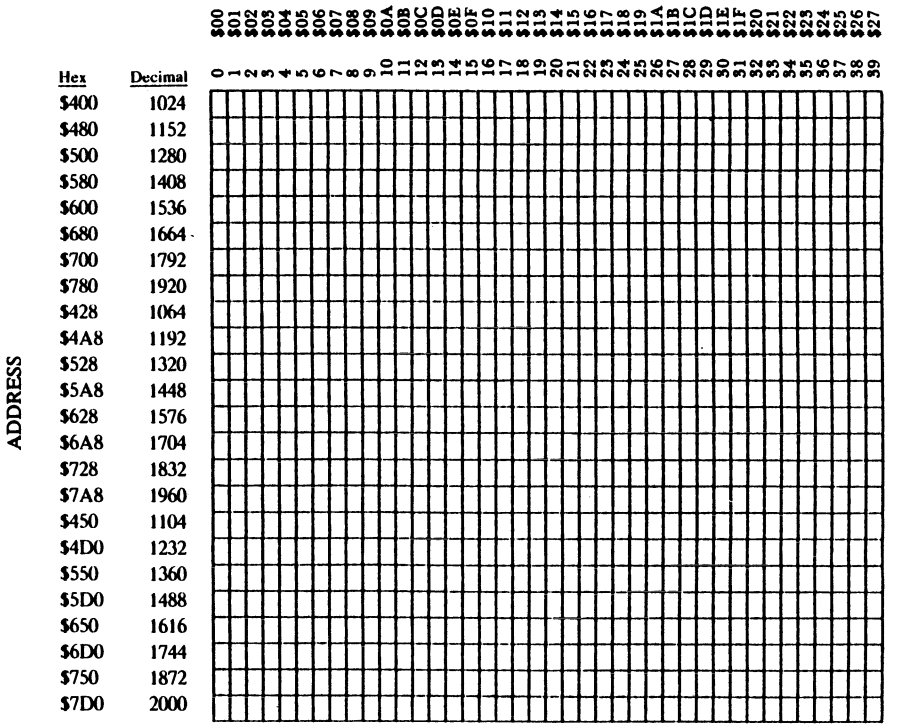


Figure A1.1 The Screen Memory Map

2 The 6502 and 65C02

So far throughout this book we have been concerned with the software aspects of the Apple's 6502 and 65C02, or in other words, how to program it! We could not really finish without having a glimpse at its hardware or physical features. For example, just how is it organized internally and how does it transfer data to and fro? While it is not absolutely vital to understand these features, an understanding of its design will enhance your new found knowledge.

Figure A2. 1 shows a simplified block diagram of the 6502 and 65C02's design or architecture as it is more commonly called. If you study it many of the features will be readily recognizable. There are a few exceptions though, including three buses, the address bus, the data bus, and the control bus. You may well be wondering just what is meant by bus? It is not, as you may have thought, a Greyhound bound for California or New York — it's simply a collective term for a series of wires — or tracks as they are called on a Printed Circuit Board (PCB for short) — onto which a 1 or a 0 can be placed electronically.

By placing a series of 1s and 0s onto the eight lines of the data bus, a byte of information may be transferred to or from the address specified by the binary value present at that instant in time on the 16 lines of the address bus.

The control bus lines are responsible for carrying the numerous synchronization signals that are required for the Apple // family of computers to operate.

EXECUTING INSTRUCTIONS

We can now examine just how the 6502 and 65C02 microprocessors fetch, interpret and execute each instruction. Firstly, they must locate and read the next instruction of the machine code program. The microprocessor — be it a 6502 or a 65C02 — does this by placing the current contents of the Program Counter onto the address bus and simultaneously placing a read signal on the appropriate control bus line.

Almost instantaneously the instruction, or more correctly the byte that constitutes the instruction, is placed onto the data bus. The microprocessor then reads the contents of the data bus into a special internal, eight bit register, known as the Instruction Register (IR for short), which is used exclusively by the 6502 or 65C02 to hold data waiting for processing. Once in the IR, the Control Unit interprets the instruction and then generates the various internal and external signals required to execute the instruction. For example, if the data byte fetched was $A5, the microprocessor would interpret this as LDA zero page, and would fetch the next byte of data and interpret this as the address at which the data to be placed into the accumulator is located. Each one of these operations would be performed in a manner similar to that already described.

Obviously instructions and data must be fetched in the correct sequence. To enable this to happen the Program Counter is provided with an automatic incrementing device. Each time the Program Counter's contents are placed onto the address bus the incrementer adds one to its contents, thus ensuring bytes are fetched and stored in the correct order.

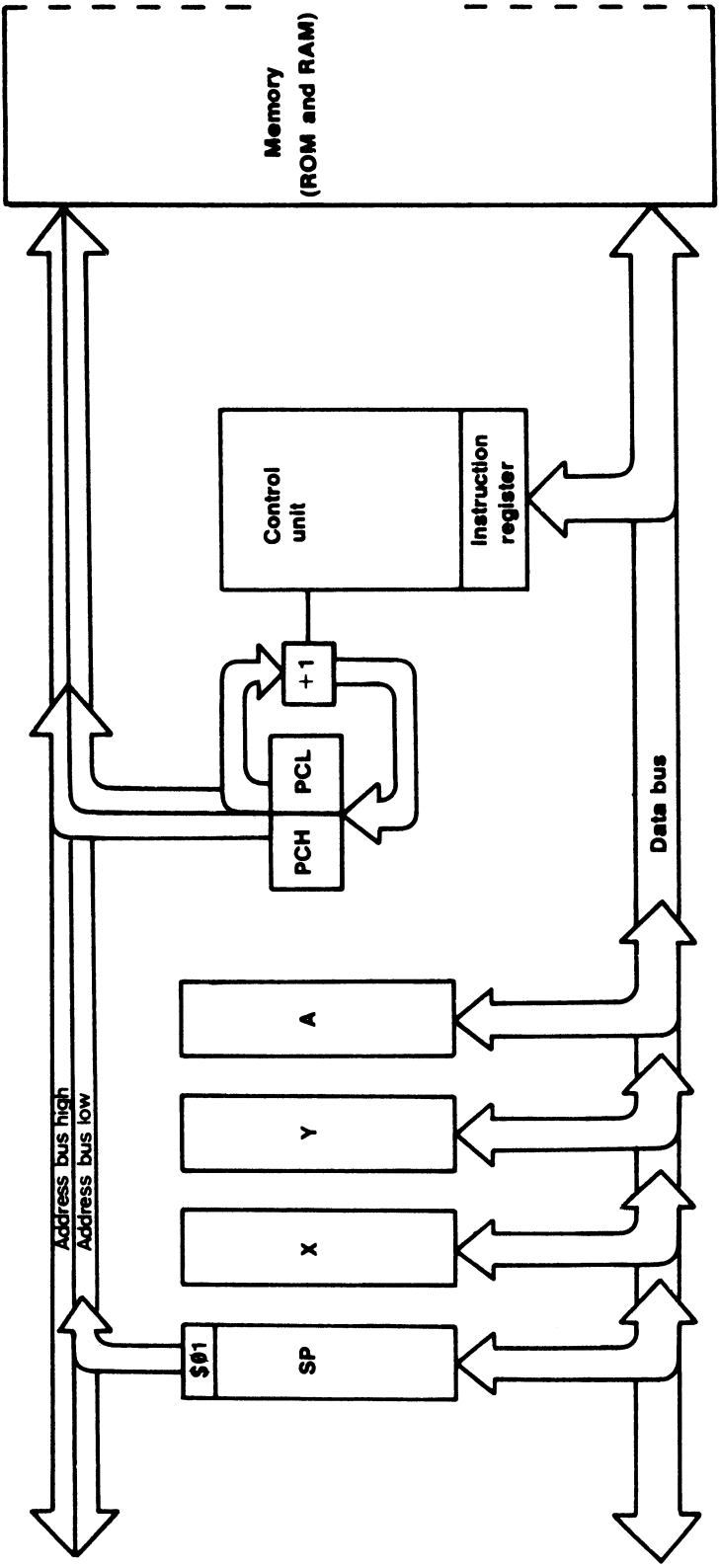


Figure A2.1 The 6502 and 65C02 block diagram.

3 The Instruction Set

This section contains a full description of each of the 64 instructions that the 6502 and 65C02 are provided with. Instructions or addressing modes that are specific to the 65C02 will be marked with an asterisk (*). For ease of reference, the instructions are arranged in alphabetical order by mnemonic, and each description is broken down into the following six sections:

Introduction: A brief one or two line description of the instruction's function.

Table: This details the addressing modes available with the instruction, and lists the various opcodes, the total number of memory bytes required by each addressing mode, and finally the number of cycles that particular addressing mode takes to complete.

Status: Shows the effect the execution of the instruction has on the Status register. The following codes are employed:

* The flag is affected by the instruction but bits are undefined, being dependent on the byte's contents
1 The flag is set by the instruction
0 The flag is cleared by the instruction

If no code is indicated the flag remains unaltered by the instruction.

Operation: A brief description of how the instruction operates together with details of its effect on the Status register.

Applications: Some hints and tips on the sort of applications the instruction might be used for.

References: A list of the page numbers giving further information about the instruction.

ADC

Add memory to accumulator with carry.

Addressing	Opcode Decimal	Opcode Hex	Bytes	Cycles
ADC #immediate	105	$69	2	2
ADC zero page	101	$65	2	3
ADC zero page, X	117	$75	2	4
ADC absolute	109	$6D	3	4
ADC absolute, X	125	$7D	4	4/5
ADC absolute, Y	121	$79	3	4/5
ADC (zero page, X)	97	$61	2	6
ADC (zero page), Y	113	$71	2	5/6

N	V	—	B	D	I	Z	C
*	*					*	*

Operation: Adds the contents of the specified memory location to the current contents of the accumulator. If the Carry flag is set this is added to the result which is then stored in the accumulator. If the result is greater than $FF (255) the Carry flag is set. If the result is equal to zero the Zero flag is set. The contents of bit 7 of the accumulator are copied into the Status register. If overflow occurred from bit 6 to bit 7 the Overflow flag is set.

Applications: Allows single, double and multibyte numbers to be added together. Overflow from one byte to another is provided by the Carry flag which is included in the addition.

References: Page 43

AND

Logical AND of memory location with accumulator.

Addressing	Opcode Decimal	Opcode Hex	Bytes	Cycles
AND #immediate	41	$29	2	2
AND zero page	37	$25	2	3
AND zero page, X	53	$35	2	4
AND absolute	45	$2D	3	4
AND absolute, X	61	$3D	3	4/5
AND absolute, Y	57	$39	3	4/5
AND (zero page, X)	33	$21	2	6
AND (zero page), Y	49	$31	2	5

N	V	—	B	D	I	Z	C
*						*	

Operation: Logically ANDs the corresponding bits of the accumulator with the specified value or contents of memory location. The result of the operation is stored in the accumulator but memory contents remain unaltered. If the result of the AND is 0, the Zero flag is set. If the result leaves bit 7 set, the Negative flag is set. Otherwise both flags are cleared.

Applications: Used to 'mask off' the unwanted bits of the accumulator.

```
AND #$F0        \ masks off lower nibble, 11110000
AND #$0F        \ masks off higher nibble, 00001111
```

References: Page 17

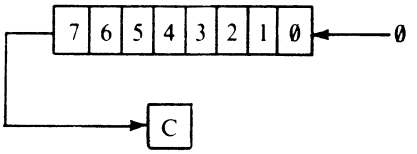
ASL

Shift contents of accumulator or memory left by one bit.

Addressing	Opcode Decimal	Opcode Hex	Bytes	Cycles
ASL accumulator	10	$0A	1	2
ASL zero page	6	$06	2	5
ASL zero page, X	22	$16	2	6
ASL absolute	14	$0E	3	6
ASL absolute, X	30	$1E	3	7

N	V	—	B	D	I	Z	C
*						*	*

Operation: Shuffles the bits in a specified location one bit left. Bit 7 moves into the carry, and a zero is placed into the vacated bit 0.



The Carry flag is set if bit 7 contained a 1 before the shift, and cleared if it contained 0. The Negative flag is set if bit 6 previously contained a 1. The Zero flag is set if the location holds $00 after the shift. (For this to occur it must previously have contained either $00 or $80.

Applications: Multiplies the byte by two. Can be used to shift low nibble of byte into high nibble.

References: Page 81

BCC

Branch if the Carry flag is clear (C = 0).

Addressing	Opcode Decimal	Hex	Bytes	Cycles
BCC relative	144	$90	2	2/3/4

N	V	—	B	D	I	Z	C

Operation: If the Carry flag is clear (C = 0) the byte following the instruction is interpreted as a two's complement number and is added to the current contents of the Program Counter. This gives the new address from which the program will now execute, allowing a branch of either 126 bytes back or 129 bytes forward. If the Carry flag is set (C = 1) the branch does not occur and the next byte is ignored by the microprocessor.

Applications: The Carry flag is conditioned by a number of instructions such as ADC, SBC, CMP, CPX and CPY, and a branch will occur if any of these result in clearing the flag. A 'forced' branch can be implemented using:

```
CLC             \ C=0
BCC value       \ 'jump'
```

References: Pages 34, 67

BCS

Branch if the Carry flag is set (C = 1).

Addressing	Opcode Decimal	Hex	Bytes	Cycles
BCS relative	176	$B0	2	2/3/4

N	V	—	B	D	I	Z	C

Operation: If the Carry flag is set (C = 1) the byte following the instruction is interpreted as a two's complement number and is added to the current contents of the Program Counter; this gives the new address from which the program will now execute, allowing a branch of either 126 bytes back or 129 bytes forward. If the Carry flag is clear (C =0) the branch does not occur and the next byte is ignored by the microprocessor.

Applications: As with BCC but the branch will only take place if an operation results in the Carry flag being set. A 'forced' branch can be implemented with:

```
SEC                    \ C = 1
BCS set                \ 'jump'
```

References: Page 34, 67

BEQ

Branch if the Zero flag is set (Z = 1).

Addressing	Opcode Decimal	Opcode Hex	Bytes	Cycles
BEQ relative	240	$F0	2	2/3/4

N	V	—	B	D	I	Z	C

Operation: If the Zero flag is set (Z = 1) the byte following the instruction is interpreted as a two's complement number and is added to the current contents of the Program Counter; this gives the new program address from which the program will now execute, allowing a branch of either 126 bytes back or 129 bytes forward. If the Zero flag is clear (Z = 0) the branch does not occur and the next byte is ignored by the microprocessor.

Applications: Used to cause a branch when the Zero flag is set. This happens when an operation results in zero (e.g. LDA #0). The BEQ command is used frequently after a comparison instruction, for example:

```
CMP #ASC"?"
BEQ Questionmark
```

If the comparison succeeds the Zero flag is set therefore BEQ will work.

References: Page 67

BIT

Test memory bits.

Addressing	Opcode Decimal	Hex	Bytes	Cycles
BIT zero page	36	$24	2	3
BIT absolute	44	$2C	3	4
*BIT zero page,X	52	$34	2	4
*BIT absolute,X	60	$3C	3	4
*BIT immediate	137	$89	2	2

N	V	—	B	D	I	Z	C
*	*					*	

Operation: The BIT operation affects only the Status register, the accumulator and the specified memory location are unaltered. Bit 7 and bit 6 of the memory byte are copied directly into N and V respectively. The Zero flag is conditioned after a logical bitwise AND between the accumulator and the memory byte. If accumulator AND memory results in zero then Z = 1, otherwise Z = 0.

Applications: Often used in conjunction with BPL/BMI or BVS/BVC to test bits 7 and 6 of a memory location and to cause a branch depending on their condition.

References: Page 88

BMI

Branch if the Negative flag is set (N = 1).

Addressing	Opcode Decimal	Opcode Hex	Bytes	Cycles
BMI relative	48	$30	2	2/3/4

N	V	—	B	D	I	Z	C

Operation: If the Negative flag is set (N = 1) the byte following the instruction is interpreted as a two's complement number and is added to the current contents of the Program Counter; this gives the new address from which the program will now execute, allowing a branch of either 126 bytes back or 129 bytes forward. If the Negative flag is clear (N = 0) the branch does not occur and the next byte is ignored by the microprocessor.

Applications: Generally after an operation has been performed (i.e. LDA, LDX etc.) the most significant bit of the register is copied into the Negative flag position. If it is set then a branch will occur using BMI. The "minus" part of the mnemonic denotes this instruction's importance when using signed arithmetic—where bit 7 is used to denote the sign of a number in two's complement form.

References: Pages 32, 68

BNE

Branch if the Zero flag is clear (Z = 0).

Addressing	Opcode Decimal	Opcode Hex	Bytes	Cycles
BNE relative	208	$D0	2	2/3/4

N	V	—	B	D	I	Z	C

Operation: If the Zero flag is clear (Z = 0) the byte following the instruction is interpreted as a two's complement number and is added to the current contents of the Program Counter; this gives the new address from which the program will now execute, allowing a branch of either 126 bytes back or 129 bytes forward. If the Zero flag is set (Z = 1) the branch does not occur and the next byte is ignored by the microprocessor.

Applications: Used to cause a branch when the Zero flag is clear. It's often used, in conjunction with a decrementing counter, as a loop controlling command.

```
DEX
BNE AGAIN
```

will continue branching back to AGAIN until X = 0 and the Zero flag is set.

References: Page 67

BPL

Branch if the Negative flag is clear (N = 0).

Addressing	Opcode Decimal	Opcode Hex	Bytes	Cycles
BPL relative	16	$10	2	3/4/5

Operation: If the Negative flag is clear (N = 0) the byte following the instruction is interpreted as a two's complement number and is added to the current contents of the Program Counter; this gives the new address from which the program will now execute, allowing a branch of either 126 bytes back or 129 bytes forward. If the Negative flag is set (N = 1) the branch does not occur and the next byte is ignored by the microprocessor.

Applications: Generally after an operation has been performed (i.e. LDA, ROL, CPX etc.) the most significant bit of the register is copied into the Negative flag position. If it is clear then a branch will occur if BPL is used. The 'plus' part of the mnemonic denotes the instruction's importance when using signed arithmetic, where bit 7 is used to indicate the sign of a number in two's complement form. If a decrementing counter is being used in a loop this branch instruction allows the loop to execute when the counter reaches zero.

```
DEX
BPL again
```

This loop will finish when X is decremented from 0 to $FF because $FF = 1111 1111 binary, where bit 7 is set.

References: Pages 32, 68

*BRA

Unconditional branch.

Addressing	Opcode Decimal	Hex	Bytes	Cycles
BRA relative	128	$80	2	2

N	V	—	B	D	I	Z	C

Operation: The byte following the instruction is interpreted as a two's complement number and is added to the current contents of the Program Counter; this gives the new address from which the program will now execute, allowing a branch of either 126 bytes back or 129 bytes forward.

Applications: Transfers control, unconditionally, to another part of the program stored close by in memory.

References: Page 68

BRK

Software forced BREAK.

Addressing	Opcode Decimal	Hex	Bytes	Cycles
BRK implied	00	$00	1	7

N	V	—	B	D	I	Z	C
			1				

Operation: The Program Counter address plus one is pushed onto the stack, followed by the contents of the Status register. The Break flag is set and the Apple passes control to the BRK servicing routine at $FFFE.

Applications: Used as a software interrupt.

References: Page 111

BVC

Branch if the Overflow flag is clear (V = 0).

Addressing	Opcode Decimal	Hex	Bytes	Cycles
BVC relative	80	$50	2	2/3/4

N	V	—	B	D	I	Z	C

Operation: If the Overflow flag is clear (V = 0) the byte following the instruction is interpreted as a two's complement number and added to the current contents of the Program Counter. This gives the new address from which the program will now execute. This allows a branch of either 126 bytes back or 129 bytes forward. If the Overflow flag is set (V = 1) the branch does not take place and the next byte is ignored by the microprocessor.

Applications: Used to detect an overflow from bit 6 into bit 7 (i.e. a carry from bit 6 to bit 7) when using signed arithmetic. When using signed arithmetic two numbers of opposite sign cannot overflow, however numbers of the same sign can overflow. For example:

```
  01001111      ($4F)
+ 01000000      ($40)
  ----------
  10001111      (-$71)
  ↑
  └── Overflow from bit 6 to bit 7
```

The result is now negative which is, of course, absurd! Similarly adding two large negative numbers can produce a positive result. In fact overflow can occur in the following situations:

1. Adding large positive numbers.
2. Adding large negative numbers.
3. Subtracting a large negative number from a large positive number.
4. Subtracting a large positive number from a large negative number.

The Overflow flag is used to signal this overflow from bit 6 to bit 7 and therefore, in signed arithmetic, a change in sign. If it is clear, no overflow has occurred and BVC will cause a branch. A 'forced' branch may be implemented using:

```
CLV                     \ clear V
BVC Forced              \ 'jump'
```

References: Page 68

BVS

Branch if the Overflow flag is set (V = 1).

Addressing	Opcode Decimal	Opcode Hex	Bytes	Cycles
BVS relative	112	$70	2	2/3/4

N	V	—	B	D	I	Z	C

Operation: If the Overflow flag is set (V = 1) the byte following the instruction is interpreted as a two's complement number and added to the current contents of the Program Counter. This gives the new address from which the program will now execute, allowing a branch of either 126 bytes back or 129 bytes forward. If the Overflow flag is clear (V = 0) the branch does not occur and the next byte is ignored by the microprocessor.

Applications: Used to cause a branch if the sign of a number has been changed. In most instances this will only matter if signed arithmetic is being employed. See BVC for more details.

References: Page 68

CLC

Clear the Carry flag (C = 0).

Addressing	Opcode Decimal	Opcode Hex	Bytes	Cycles
CLC implied	24	$18	1	2

N	V	—	B	D	I	Z	C
							0

Operation: The Carry flag is cleared by setting it to zero.

Applications: Should always be used before adding two numbers together as the Carry flag's contents are taken into account by ADC. A 'forced' branch may be implemented with:

```
CLC          \ Clear C
BCC clear    \ and 'jump'
```

References: Pages 34, 42

CLD

Clear the Decimal flag (D = 0).

Addressing	Opcode Decimal	Opcode Hex	Bytes	Cycles
CLD implied	216	$D8	1	2

N	V	—	B	D	I	Z	C
				0			

Operation: The Decimal flag is cleared by setting it to zero.

Applications: Used to make microprocessor work in normal hexadecimal mode.

References: Page 49

CLI

Clear the Interrupt flag (1 = 0).

Addressing	Opcode Decimal	Opcode Hex	Bytes	Cycles
CLI implied	88	$58	1	2

N	V	—	B	D	I	Z	C
					0		

Operation: The Interrupt flag is cleared by setting it to zero.

Applications: Causes any interrupts on the IRQ line to be processed immediately after completion of current instruction.

References: Page 110

CLV

Clear the Overflow flag (V = 0).

Addressing	Opcode Decimal	Opcode Hex	Bytes	Cycles
CLV implied	184	$B8	1	2

Operation: The Overflow flag is cleared by setting it to zero.

Applications: Used to clear the Overflow flag after an overflow from bit 6 to bit 7. In most instances this is only important if signed arithmetic is being used.

CMP

Compare contents of memory with contents of the accumulator.

Addressing	Opcode Decimal	Opcode Hex	Bytes	Cycles
CMP #immediate	201	$C9	2	2
CMP zero page	197	$C5	2	3
CMP zero page,X	213	$D5	2	4
CMP absolute	205	$CD	3	4
CMP absolute,X	221	$DD	3	4/5
CMP absolute,Y	217	$D9	3	4/5
CMP (zero page,X)	193	$C1	2	6
CMP (zero page),Y	209	$D1	2	5/6
*CMP (indirect)	210	$D2	2	5

N	V	—	B	D	I	Z	C
*						*	*

Operation: The contents of the specified memory location (or immediate value) are subtracted from the contents of the accumulator. The contents of the memory location and accumulator are NOT altered, but the Negative, Zero and Carry flags are conditioned according to the result of the subtraction. To perform this subtraction, the microprocessor first sets the Carry flag and then adds the two's complement value of the memory location's contents to the accumulator's contents. If both values are equal (memory = accumulator) the Zero flag is set and the Carry flag remains set. If the contents of memory are less than the accumulator (memory < accumulator) the Zero flag is cleared and the Carry flag set. If memory contents are greater than the accumulator (memory > accumulator) then both the Zero flag and Carry flag are cleared. If unsigned binary is being used the Negative flag is also set. If signed binary is being used the Overflow flag should be checked in conjunction with the Negative flag to test for a 'true' negative result.

Applications: Should be used to test for intermediate values that cannot be tested directly from the Status register. For example:

```
CMP #00
BEQ AWAY
```

is a waste of two bytes, as the Zero flag will be set if the accumulator contains $00, therefore all that is needed is : BEQ AWAY. To test for a particular key, the following CMP might be used:

```
JSR GETIN          \ get key
CMP #ASC"Y"        \ is it Y key?
BEQ YES
```

References: Page 67

CPX

Compare contents of memory with contents of the X register.

Addressing	Opcode Decimal	Opcode Hex	Bytes	Cycles
CPX #immediate	224	$E0	2	2
CPX zero page	228	$E4	2	3
CPX absolute	236	$EC	3	4

N	V	—	B	D	I	Z	C
*						*	*

Operation: The contents of the specified memory location (or immediate value) are subtracted from the contents of the X register. The contents of the memory location and X register are NOT altered, instead the Negative, Zero and Carry flags are conditioned according to the result of the subtraction. To perform this subtraction the microprocessor first sets the Carry flag and then adds the two's complement value of the memory location to the contents of the X register. If both values are equal (memory = X register) the Zero flag is set and the Carry flag remains set. If the contents of memory are less than the X register (memory < X register) the Zero flag is cleared but the Carry flag remains set. If memory contents are greater than the X register (memory > X register) then both Zero and Carry flags are cleared. If unsigned binary is being used then the Negative flag is set.

Applications: Should be used to test for intermediate values which cannot be tested directly from the Status register. For example, to test the X register's contents during use as a loop counter try:

```
        LDX #220        \ load X with 220
AGAIN   DEX             \ decrement X
        CPX #87         \ has X reached 87?
        BNE AGAIN       \ no, go again
```

References: Page 67

CPY

Compare contents of memory with contents of the Y register.

Addressing	Opcode Decimal	Hex	Bytes	Cycles
CPY #immediate	192	$C0	2	2
CPY zero page	196	$C4	2	3
CPY absolute	204	$CC	3	4

N	V	—	B	D	I	Z	C
*						*	*

Operation: The contents of the specified memory location (or immediate value) are subtracted from the contents of the Y register. The contents of the memory location and Y register are NOT altered, instead the Negative, Zero and Carry flags are conditioned according to the result of the subtraction. To perform this subtraction the microprocessor first sets the Carry flag and then adds the two's complement value of the memory location to the contents of the Y register. If both values are equal (memory = Y register) the Zero flag is set and the Carry flag remains set. If the contents of memory are less than the Y register (memory < Y register) the Zero flag is cleared but the Carry flag remains set. If memory contents are greater than the Y register (memory > Y register) then both Zero and Carry flags are cleared. If unsigned binary is being used then the Negative flag is set.

Applications: Should be used to test for intermediate values which cannot be tested directly from the Status register. For example, to test the Y register's contents during use as a loop counter try:

```
        LDY #220        \ load Y with 220
AGAIN   DEY             \ decrement Y
        CPY #87         \ has Y reached 87?
        BNE AGAIN       \ no, go again
```

References: Page 67

DEC

Decrement memory contents by one.

Addressing	Opcode Decimal	Hex	Bytes	Cycles
DEC zero page	198	$C6	2	5
DEC zero page,X	214	$D6	2	6
*DEC implied	58	$3A	1	2
DEC absolute	206	$CE	3	6
DEC absolute,X	222	$DE	3	7

N	V	—	B	D	I	Z	C
*						*	

Operation: The byte at the address specified is decremented by one (MEMORY = MEMORY -1). If the result of the operation is zero the Zero flag will be set. Bit 7 of the byte is copied into the Negative flag.

Applications: Used to subtract one from a counter stored in memory.

References: Page 66

DEX

Decrement contents of X register by one.

Addressing	Opcode Decimal	Opcode Hex	Bytes	Cycles
DEX implied	202	$CA	1	2

N	V	—	B	D	I	Z	C
*						*	

Operation: One is subtracted from the value currently held in the X register (X= X -1). If the result of the operation is zero the Zero flag will be set. Bit 7 is copied into the Negative flag (N = 0 if X < $80; N = 1 if X > $7F). The Carry flag is not affected by the instruction.

Applications: Used with indexed addressing when the X register acts as an offset from a base address, allowing a sequential set of bytes to be accessed. Invariably used to decrement the X register when being used as a loop counter, branching until X = 0 (Z = 1).

References: Page 65

DEY

Decrement contents of Y register by one.

Addressing	Opcode Decimal	Opcode Hex	Bytes	Cycles
DEY implied	136	$88	1	2

N	V	—	B	D	Z	C
*					*	

Operation: One is subtracted from the value currently held in the Y register (Y = Y -1). If the result of the operation is zero the Zero flag is set. Bit 7 is copied into the Negative flag (N = 0 if Y < $80;N = 1 if Y > $7F). The Carry flag is not affected by the instruction.

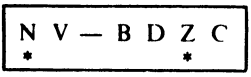
Applications: Used with indexed addressing when the Y register acts as an offset from a base address allowing a sequential set of bytes to be accessed. Invariably used to decrement the Y register when being used as a loop counter, branching until Y = 0 (Z = 1).

References: Page 65

EOR

Accumulator exclusively ORed with memory.

Addressing	Opcode Decimal	Hex	Bytes	Cycles
EOR #immediate	73	$49	2	2
EOR zero page	69	$45	2	3
EOR zero page,X	85	$55	2	4/5
EOR absolute	77	$4D	3	4
EOR absolute,X	93	$5D	3	4/5
EOR absolute,Y	89	$59	3	4/5
EOR (zero page,X)	65	$41	2	6
EOR (zero page),Y	81	$51	2	5/6
*EOR (indirect)	82	$52	2	5



Operation: Performs a bitwise exclusive OR between the corresponding bits in the accumulator and the specified memory byte. If the result, which is stored in the accumulator, is zero the Zero flag is set. Bit 7 is copied into the Negative flag.

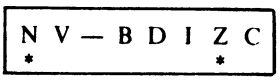
Applications: Used to complement or invert a data byte.

References: Page 18

INC

Increment memory contents by one.

Addressing	Opcode Decimal	Hex	Bytes	Cycles
INC zero page	230	$E6	2	5
INC zero page,X	246	$F6	2	6
*INC implied	26	$1A	1	2
INC absolute	238	$EE	3	6
INC absolute,X	254	$FE	3	7



Operation: The byte at the address specified is incremented by one. If the address holds zero after the operation the Zero flag is set. Bit 7 of the byte is copied into the Negative flag.

Applications: Add one to a counter stored in memory.

References: Page 66

INX

Increment contents of X register by one.

Addressing	Opcode Decimal	Opcode Hex	Bytes	Cycles
INX implied	232	$E8	1	2

N	V	—	B	D	I	Z	C
*						*	

Operation: One is added to the value currently in the X register (X = X + 1). If the result of the operation is zero the Zero flag will be set. Bit 7 is copied into the Negative flag (N = 0 if X < $80 ; N = 1 if X > $7F). The Carry flag is not affected by the instruction.

Applications: Used with indexed addressing when the X register acts as an offset from a base address, and allows a sequential set of bytes to be accessed. Often used as a counter to control the number of times a loop of instructions is executed.

References: Page 65

INY

Increment contents of Y register by one.

Addressing	Opcode Decimal	Opcode Hex	Bytes	Cycles
INY implied	200	$C8	1	2

N	V	—	B	D	I	Z	C
*						*	

Operation: One is added to the value currently held in the Y register (Y = Y + 1). If the result of the operation is zero the Zero flag will be set. Bit 7 is copied into the Negative flag. The Carry flag is not affected.

Applications: Used with indexed addressing when the Y register acts as an offset from a base address, allowing a sequential set of bytes to be accessed. Often used as a counter to control the number of times a loop is executed.

References: Page 65

JMP

Jump to a new location.

Addressing	Opcode Decimal	Hex	Bytes	Cycles
JMP absolute	76	$4C	3	3
*JMP (zero page,X)	124	$7C	3	6
JMP (indirect)	108	$6C	3	3

N	V	—	B	D	I	Z	C

Operation: In an absolute JMP the two bytes following the instruction are placed into the Program Counter. In an indirect jump the two bytes located at the two byte address following the instruction are loaded into the Program Counter.

Applications: Transfers control, unconditionally, to another part of a program stored anywhere in memory.

References: Pages 56, 79

JSR

Jump, save return address.

Addressing	Opcode Decimal	Hex	Bytes	Cycles
JSR absolute	32	$20	3	6

N	V	—	B	D	I	Z	C

Operation: Acts as a subroutine call, transferring program control to another part of memory until an RTS is encountered. The current contents of the Program Counter plus two are pushed onto the stack. The Stack Pointer is incremented twice. The absolute address following the instruction is placed into the Program Counter and program execution continues from this new address.

Applications: Allows large repetitive sections of programs to be entered once, out of the way of the main program, and called as subroutines as often as required.

References: Page 75

LDA

Load the accumulator with the specified byte.

Addressing	Opcode Decimal	Hex	Bytes	Cycles
LDA #immediate	169	$A9	2	2
LDA zero page	165	$A5	2	3
LDA zero page,X	181	$B5	2	4
LDA absolute	173	$AD	3	4
LDA absolute,X	189	$BD	3	4/5
LDA absolute,Y	185	$B9	3	4/5
LDA (zero page,X)	161	$A1	2	6
LDA (zero page),Y	177	$B1	2	5/6
*LDA (indirect)	178	$B2	2	5

N	V	—	B	D	I	Z	C
*						*	

Operation: Places the value immediately following the instruction, or the contents of the location specified after the instruction, into the accumulator. If the value loaded is zero then the Zero flag is set. Bit 7 is copied into the Negative flag position.

Applications: Probably the most frequently used instruction, it allows for general data movement and facilitates all logical and arithmetic operations.

References: Page 38

LDX

Load the X register with the specified byte.

Addressing	Opcode Decimal	Hex	Bytes	Cycles
LDX #immediate	162	$A2	2	2
LDX zero page	166	$A6	2	3
LDX zero page, Y	182	$B6	2	4
LDX absolute	174	$AE	3	4
LDX absolute, Y	190	$BE	3	4/5

N	V	—	B	D	I	Z	C
*						*	

Operation: Places the value immediately following the instruction, or the contents of the location specified after the instruction, into the X register. If the value loaded is zero then the Zero flag is set. Bit 7 is copied into the Negative flag position.

Applications: General transfer of data for processing or storage. Also allows a loop counter to be set to its start value.

References: Page 38

LDY

Load the Y register with the specified byte.

Addressing	Opcode Decimal	Opcode Hex	Bytes	Cycles
LDY #immediate	160	$A0	2	2
LDY zero page	164	$A4	2	3
LDY zero page, X	180	$B4	2	4
LDY absolute	172	$AC	3	4
LDY absolute, X	188	$BC	3	4/5

N	V	—	B	D	I	Z	C
*						*	

Operation: Places the value immediately following the instruction, or the contents of the location specified after the instruction, into the Y register. If the value loaded is zero the Zero flag is set. Bit 7 is copied into the Negative flag.

Applications: General transfer of data for processing or storage. Also allows a loop counter to be set to its start value.

References: Page 38

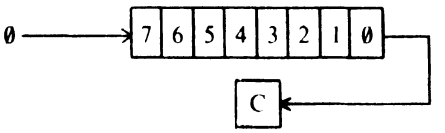
LSR

Logically shift the specified byte right one bit.

Addressing	Opcode Decimal	Hex	Bytes	Cycles
LSR accumulator	74	$4A	1	2
LSR zero page	70	$46	2	5
LSR zero page, X	86	$56	2	6
LSR absolute	78	$4E	3	6
LSR absolute, X	94	$5E	3	7

N	V	—	B	D	I	Z	C
0						*	*

Operation: Moves the contents of the specified byte right by one position, putting a 0 in bit 7 and bit 0 into the Carry flag.



The Negative flag is cleared, and the Carry flag is conditioned by the contents of bit 0. The Zero flag is set if the specified byte now holds zero (in which case it must previously have contained $00 or $01).

Applications: Divides a byte value by two (if D = 0) with its remainder shifting into the Carry flag position. Can also be used to shift the high nibble of a byte into the low nibble.

References: Pages 82, 83

NOP

No operation.

Addressing	Opcode Decimal	Hex	Bytes	Cycles
NOP implied	234	$EA	1	2

N	V	—	B	D	I	Z	C

Operation: Does nothing except increment the Program Counter.

Applications: Provides a two cycle delay.

References: Page 109

ORA

Logical OR of a specified byte with the accumulator.

Addressing	Opcode Decimal	Opcode Hex	Bytes	Cycles
ORA #immediate	9	$09	2	2
ORA zero page	5	$05	2	3
ORA zero page,X	21	$15	2	4
ORA absolute	13	$0D	3	4
ORA absolute,X	29	$1D	3	4/5
ORA absolute,Y	25	$19	3	4/5
ORA (zero page,X)	1	$01	2	6
ORA (zero page),Y	17	$11	2	5
*ORA (indirect)	18	$12	2	5

N	V	—	B	D	I	Z	C
*						*	

Operation: Logically ORs the corresponding bits of the accumulator with the specified value, or contents of a memory location. The result of the operation is stored in the accumulator. If the result leaves bit 7 set the Negative flag is set, otherwise it is cleared.

Applications: Used to 'force' certain bits to contain a one. For example:

```
ORA #$80                    \  10000000 binary
```

will ensure bit 7 is set.

References: Pages 52, 53, 55

PHA

Push the accumulator contents onto the 'top' of the stack.

Addressing	Opcode Decimal	Opcode Hex	Bytes	Cycles
PHA implied	72	$48	1	3

N	V	—	B	D	I	Z	C

Operation: The contents of the accumulator are copied into the position indicated by the Stack Pointer. The Stack Pointer is then decremented by one.

Applications: Allows bytes of memory to be saved temporarily. The index registers can be saved by first transferring them to the accumulator; memory bytes are saved by first loading them into the accumulator. Bytes are recovered with PLA.

References: Pages 61, 62

PHP

Push the Status register's contents onto the top of the stack.

Addressing	Opcode Decimal	Opcode Hex	Bytes	Cycles
PHP implied	8	$08	1	3

N	V	—	B	D	I	Z	C

Operation: The contents of the Status register are copied into the position indicated by the Stack Pointer. The Stack Pointer is then decremented by one.

Applications: Allows the conditions of the flags to be saved, perhaps prior to a subroutine call, so that the same conditions can be restored with PLP on return.

References: Pages 61, 62

*PHX

Push the X register's contents onto the top of the stack.

Addressing	Opcode Decimal	Opcode Hex	Bytes	Cycles
PHX implied	218	$DA	1	3

N	V	—	B	D	I	Z	C

Operation: The contents of the X register are copied into the position indicated by the Stack Pointer. The Stack Pointer is then decremented by one.

Applications: Allows the contents of the X register to be saved, perhaps prior to a subroutine call, so it contents can be restored with a PLX on return.

References: Page 61

*PHY

Push the Y register's contents onto the top of the stack.

Addressing	Opcode Decimal	Opcode Hex	Bytes	Cycles
PHY implied	90	$5A	1	3

N	V	—	B	D	I	Z	C

Operation: The contents of the Y register are copied into the position indicated by the Stack Pointer. The Stack Pointer is then decremented by one.

Applications: Allows the contents of the Y register to be saved, perhaps prior to a subroutine call, so it contents can be restored with a PLY on return.

References: Page 61

PLA

Pull the 'top' of the stack into the accumulator.

Addressing	Opcode Decimal	Opcode Hex	Bytes	Cycles
PLA implied	104	$68	1	4

N	V	—	B	D	I	Z	C
*						*	

Operation: The Stack Pointer is incremented by one, and the byte contained at this position in the stack is copied into the accumulator. If the byte is $00 the Zero flag is set. Bit 7 is copied into the Negative flag.

Applications: Complements the operation of PHA to retrieve data previously pushed onto the stack.

References: Pages 61, 62

PLP

Pull the 'top' of the stack into the Status register.

Addressing	Opcode Decimal	Opcode Hex	Bytes	Cycles
PLP implied	40	$28	1	4

N	V	—	B	D	I	Z	C
*	*		*	*	*	*	*

Operation: The Stack Pointer is incremented by one and the byte contained at this position is copied into the Status register.

Applications: Complements the operation of PHP to retrieve the previously pushed contents of the Status register, or to condition certain flags from a defined byte previously pushed onto the stack via the accumulator.

References: Pages 61, 62

*PLX

Pull the top of the stack into the X register.

Addressing	Opcode Decimal	Opcode Hex	Bytes	Cycles
PLX implied	250	$FA	1	4

N	V	—	B	D	I	Z	C
*						*	

Operation: The Stack Pointer is incremented by one and the byte contained at this position is copied into the X register.

Applications: Complements the operation of PHX to retrieve the previously pushed contents of the X register, or to condition certain flags from a defined byte previously pushed onto the stack via the accumulator.

References: Page 61

*PLY

Pull the top of the stack into the Y register.

Addressing	Opcode Decimal	Opcode Hex	Bytes	Cycles
PLY implied	122	$7A	1	4

N	V	—	B	D	I	Z	C
*						*	

Operation: The Stack Pointer is incremented by one and the byte contained at this position is copied into the Y register.

Applications: Complements the operation of PHY to retrieve the previously pushed contents of the Y register, or to condition certain flags from a defined byte previously pushed onto the stack via the accumulator.

References: Page 61

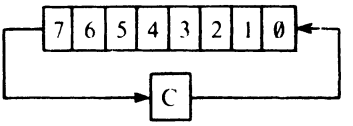
ROL

Rotate either the accumulator or a memory byte left by one bit with the Carry flag.

Addressing	Opcode Decimal	Opcode Hex	Bytes	Cycles
ROL accumulator	42	$2A	1	2
ROL zero page	38	$26	2	5
ROL zero page, X	54	$36	2	6
ROL absolute	46	$2E	3	6
ROL absolute, X	62	$3E	3	7

N	V	—	B	D	I	Z	C
*						*	*

Operation: The specified byte and the contents of the Carry flag are rotated left by one bit in a circular manner.



Bit 7 is rotated into the Carry flag, with the flag's previous contents moving into bit 0. The remaining bits are shuffled left. The Negative flag is set if bit 6 previously held 1; cleared otherwise. The Carry flag is conditioned by bit 7, and if the specified byte now holds zero the Zero flag is set.

Applications: Used in conjunction with ASL, ROL can be used to double the value of multibyte numbers, as the Carry bit is used to propagate the overflow from one byte to another. It may also be used before testing the Negative, Zero and Carry flags to determine the state of specific bits.

References: Page 81

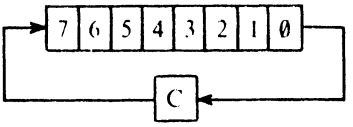
ROR

Rotate either the accumulator or a memory byte right by one bit with the Carry flag.

Addressing	Opcode Decimal	Opcode Hex	Bytes	Cycles
ROR accumulator	106	$6A	1	2
ROR zero page	102	$66	2	5
ROR zero page, X	118	$76	2	6
ROR absolute	110	$6E	3	6
ROR absolute, X	126	$7E	3	7

N	V	—	B	D	I	Z	C
*						*	*

Operation: The specified byte with the contents of the Carry flag are rotated right by one bit in a circular manner.



Bit 0 is rotated into the Carry flag with the flag's previous contents moving into the bit 7 position. The remaining bits are shuffled right. The Negative flag is set if the Carry flag was set previously; otherwise it is cleared. If bit 0 contained a 1 the Carry flag will now also be set. If the specified byte now holds zero the Zero flag is set.

Applications: Used in conjunction with LSR, ROR can be used to halve the value of multibyte numbers. It may also be used before testing the Negative, Zero and Carry flags to determine the contents of specific bits.

References: Pages 81, 82

RTI

Return from interrupt.

Addressing	Opcode Decimal	Opcode Hex	Bytes	Cycles
RTI implied	64	$40	1	6

N	V	—	B	D	I	Z	C
*	*		*	*	*	*	*

Operation: This instruction expects to find three bytes on the stack. The first byte is pulled from the stack and placed into the Status register—thus conditioning all flags. The next two bytes are placed into the Program Counter. The Stack Pointer is incremented as each byte is pulled.

Applications: Used to restore control to a program after an interrupt has occurred. On detecting the interrupt, the processor will have pushed the Program Counter and Status register onto the stack.

References: Page 111

RTS

Return from subroutine.

Addressing	Opcode Decimal	Opcode Hex	Bytes	Cycles
RTS implied	96	$60	1	6

N	V	—	B	D	I	Z	C

Operation: The two bytes on the top of the stack are pulled, incremented by one, and placed into the Program Counter. Program execution continues from this address. The Stack Pointer is incremented by two.

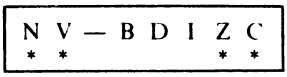
Applications: Returns control from a subroutine to the calling program. It should therefore be the last instruction of a subroutine.

References: Page 75

SBC

Subtract specified byte from the accumulator with borrow.

Addressing	Opcode Decimal	Hex	Bytes	Cycles
SBC #immediate	233	$E9	2	2
SBC zero page	229	$E5	2	3
SBC zero page,X	245	$F5	2	4
SBC absolute	237	$ED	3	4
SBC absolute,X	253	$FD	3	4/5
SBC absolute,Y	247	$F9	3	4/5
SBC (zero page,X)	225	$E1	2	6
SBC (zero page),Y	241	$F1	2	5/6
*SBC (indirect)	242	$F2	2	5



Operation: Subtracts the immediate value, or the byte contained at the specified address, from the contents of the accumulator. If the value is greater than the contents of the accumulator it will 'borrow' from the Carry flag, which should be set at the onset (only) of a subtraction. If the Carry flag is clear after the subtraction, a borrow has occurred. If the result is $00 the Zero flag is set. The contents of bit 7 are copied into the accumulator and V is set if an overflow from bit 6 to bit 7 occurred.

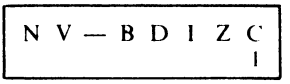
Applications: Allows single, double and multibyte numbers to be subtracted from one another.

References: Pages 45, 46

SEC

Set the Carry flag (C = 1).

Addressing	Opcode Decimal	Hex	Bytes	Cycles
SEC implied	56	$38	1	2



Operation: A one is placed into the Carry flag bit position.

Applications: Should always be used at the onset of subtraction as the Carry flag is taken into account by SBC.

References: Pages 34, 45

SED

Set the Decimal mode flag (D = 1).

Addressing	Opcode Decimal	Opcode Hex	Bytes	Cycles
SED implied	248	$F8	1	2

N	V	—	B	D	I	Z	C
				1			

Operation: A one is placed into the Decimal flag position.

Applications: Puts the Commodore in decimal mode, in which Binary Coded Decimal (BCD) arithmetic is performed. The Carry flag now denotes a carry of hundreds, as the maximum value that can be encoded in a single BCD byte is 99.

References: Page 49

SEI

Set the Interrupt disable flag (I = 1).

Addressing	Opcode Decimal	Opcode Hex	Bytes	Cycles
SEI implied	120	$78	1	2

N	V	—	B	D	I	Z	C
					1		

Operation: A one is placed into the Interrupt flag position.

Applications: When this flag is set no interrupts occurring on the IRQ line are processed. However NMI interrupts are processed, as are BREAKs.

References: Page 111

STA

Store the accumulator's contents in a memory location.

Addressing	Opcode Decimal	Hex	Bytes	Cycles
STA zero page	133	$85	2	3
STA zero page,X	149	$95	2	4
STA absolute	141	$8D	3	4
STA absolute,X	157	$9D	3	5
STA absolute,Y	137	$99	3	5
STA (zero page,X)	129	$81	2	6
STA (zero page),Y	145	$91	2	6
* STA (indirect)	145	$92	2	5

N	V	—	B	D	I	Z	C

Operation:s The contents of the accumulator are copied into the specified memory location.

Applications: To save the contents of the accumulator, or to initialize areas of memory to specific values. Used in conjunction with LDA, blocks of data can be transferred from one area of memory to another.

References: Page 39

STX

Store the X register's contents in memory.

Addressing	Opcode Decimal	Hex	Bytes	Cycles
STX zero page	134	$86	2	3
STX zero page, X	150	$96	2	4
STX absolute	142	$8E	3	4

N	V	—	B	D	I	Z	C

Operation: The contents of the X register are copied into the specified memory location.

Applications: To save the X register's contents, or to initialize areas of memory to specific values.

References: Page 39

STY

Store the Y register's contents in memory.

Addressing	Opcode Decimal	Hex	Bytes	Cycles
STY zero page	132	$84	2	3
STY zero page, X	148	$94	2	4
STY absolute	140	$8C	3	4

N	V	—	B	D	I	Z	C

Operation:s The contents of the Y register are copied into the specified memory location.

Applications: To save the Y register's contents, or to initialize areas of memory to specific values.

References: Page 39

*STZ

Store a zero in a memory location.

Addressing	Opcode Decimal	Hex	Bytes	Cycles
STZ absolute	156	$9C	3	4
STZ zero page	100	$64	2	3
STZ zero page,X	116	$74	2	4
STZ absolute,X	158	$9E	3	5

N	V	—	B	D	I	Z	C

Operation: A zero is copied into the specified memory location.

Applications: To initialize areas of memory to a zero value.

References: Page 39

TAX

Transfer the accumulator's contents into the X register.

Addressing	Opcode Decimal	Opcode Hex	Bytes	Cycles
TAX implied	170	$AA	1	2

N	V	—	B	D	I	Z	C
*						*	

Operation: The contents of the accumulator are copied into the X register. If the X register now holds zero, the Zero flag is set. Bit 7 is copied into the Negative flag.

Applications: Allows the accumulator's values to be saved temporarily, or perhaps used to seed the X register as a loop counter. Often used after PLA to restore the X register's contents previously pushed onto the stack.

References: Page 39

TAY

Transfer accumulator's contents into the Y register.

Addressing	Opcode Decimal	Opcode Hex	Bytes	Cycles
TAY implied	168	$A8	1	2

N	V	—	B	D	I	Z	C
*						*	

Operation: The contents of the accumulator are copied into the Y register. If the Y register now holds zero the Zero flag is set. Bit 7 is copied into the Negative flag.

Applications: Allows the accumulator's values to be saved temporarily, or perhaps used to seed the Y register as a loop counter. Often used after PLA to restore the Y register's contents previously pushed onto the stack.

References: Page 39

*TRB

Test and reset memory bits with the Accumulator.

Addressing	Opcode Decimal	Hex	Bytes	Cycles
TRB zero page	20	$14	2	5
TRB absolute	28	$1C	3	6

N	V	—	B	D	I	Z	C
						*	

Operation: The TRB operation affects only the the Status Register and the specified memory location, the accumulator remains unaltered. The Z flag is conditioned after a logical bitwise AND between the accumulator and the memory byte. If accumulator AND memory results in zero then Z=1, otherwise Z=0. Results of the TRB operation are stored in the specified memory byte.

Applications: Used to test a memory location against a predetermined pattern that is stored in the accumulator and reset a memory location.

References: Page 36

*TSB

Test and set memory bits with the Accumulator.

Addressing	Opcode Decimal	Hex	Bytes	Cycles
TSB zero page	4	$04	2	5
TSB absolute	12	$0C	3	6

N	V	—	B	D	I	Z	C
						*	

Operation: The TSB operation affects only the the Status Register and the specified memory location, the accumulator remains unaltered. The Z flag is conditioned after a logical bitwise OR between the accumulator and the memory byte. If accumulator OR memory results in zero then Z=1, otherwise Z=0. Results of the TSB operation are stored in the specified memory byte.

Applications: Used to test a memory location against a predetermined pattern that is stored in the accumulator and set a memory location.

References: Page 36

TSX

Transfer the Stack Pointer's contents into the X register.

Addressing	Opcode Decimal	Opcode Hex	Bytes	Cycles
TSX implied	186	$BA	1	2

N	V	—	B	D	I	Z	C
*						*	

Operation: The contents of the Stack Pointer are copied into the X register. If X now holds zero, the Zero flag is set. Bit 7 is copied into the Negative flag.

Applications: To calculate the amount of space left on the stack, or to save its current position while the stack contents are checked.

References: Page 64

TXA

Transfer the X Register's contents into the accumulator.

Addressing	Opcode Decimal	Opcode Hex	Bytes	Cycles
TXA implied	138	$8A	1	2

N	V	—	B	D	I	Z	C
*						*	

Operation: The contents of the X register are copied into the accumulator. If the accumulator now holds zero the Zero flag is set. Bit 7 is copied into the Negative flag.

Applications: Allows the X register's contents to be manipulated by logical or arithmetic instructions. Followed by a PHA it allows the X register's value to be saved on the stack.

References: Page 39

TXS

Transfer the X Register's contents into the Stack Pointer.

Addressing	Opcode Decimal	Hex	Bytes	Cycles
TXS implied	154	$9A	1	2

N	V	—	B	D	I	Z	C

Operation: The contents of the X register are copied into the Stack Pointer.

Applications: Allows the contents of the Stack Pointer to be set or reset to a specific value. For example, on. 'power-up' or BREAK, the OS executes:

```
LDX #$FF
TXS
```

to 'clear' the stack and reset the Stack Pointer.

References: Page 64

TYA

Transfer the Y register's contents into the accumulator.

Addressing	Opcode Decimal	Hex	Bytes	Cycles
TYA implied	152	$98	1	2

N	V	—	B	D	I	Z	C
*						*	

Operation: The contents of the Y register are copied into the accumulator. If the accumulator now holds zero the Zero flag is set. Bit 7 is copied into the Negative flag.

Applications: Allows the Y register's contents to be manipulated by logical or arithmetic instructions. When followed by a PHA, it allows the Y register's value to be saved on the stack.

References: Page 39

4 Instruction Cycle Times

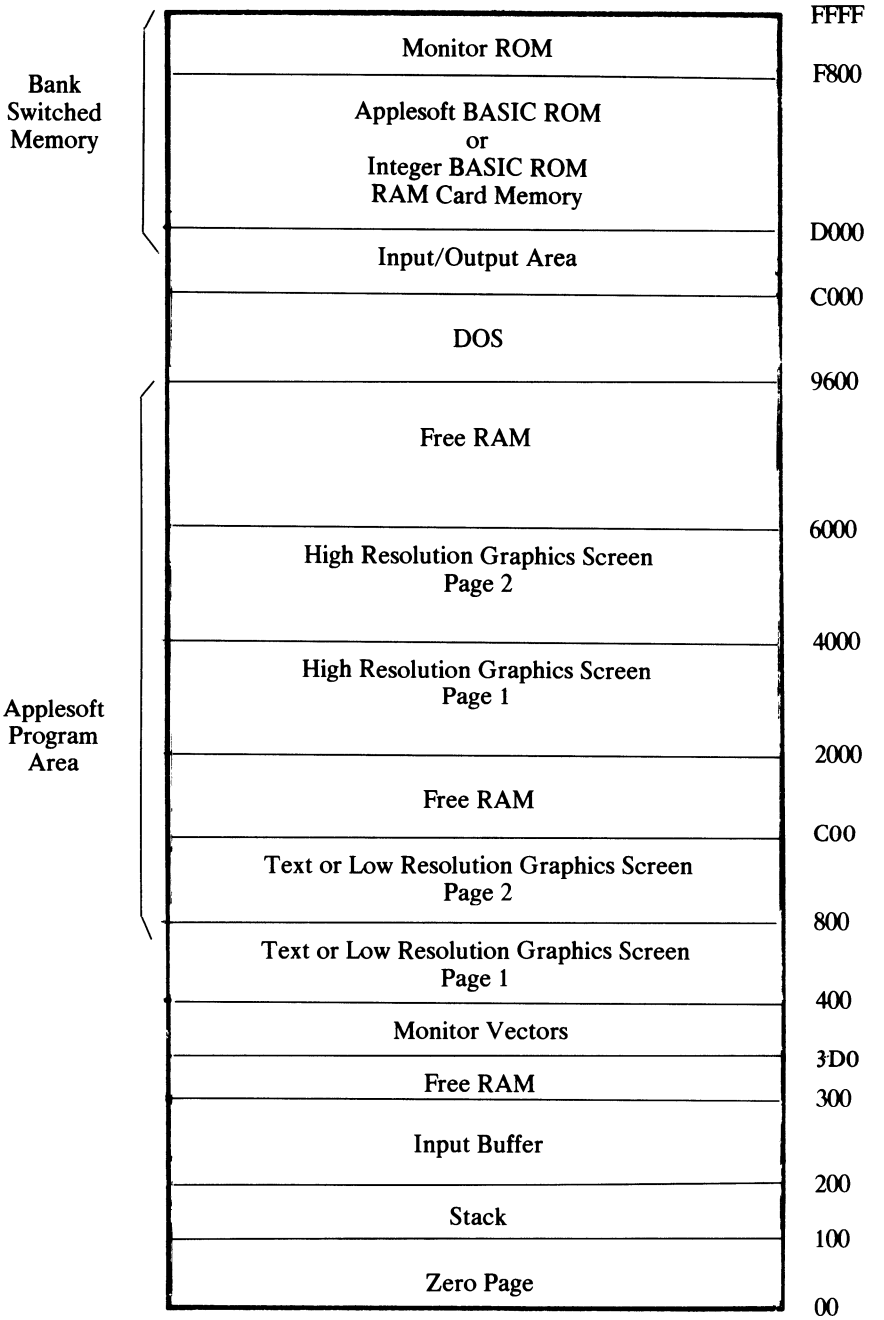
	Implied	Relative	Immediate	Zero page	Zero page, X	Absolute	Absolute, X	Absolute, Y	(Zero page, X)	(Zero page), Y	(Indirect)
ADC	—	—	2	3	4	4	4*	4*	6	5*	—
AND	—	—	2	3	4	4	4*	4*	6	5*	—
ASL	2	—	—	5	6	6	7	—	—	—	—
BCC	—	2**	—	—	—	—	—	—	—	—	—
BCS	—	2**	—	—	—	—	—	—	—	—	—
BEQ	—	2**	—	—	—	—	—	—	—	—	—
BIT	—	—	2†	3	4†	4	4†	—	—	—	—
BMI	—	2**	—	—	—	—	—	—	—	—	—
BNE	—	2**	—	—	—	—	—	—	—	—	—
BPL	—	2**	—	—	—	—	—	—	—	—	—
BRA	—	2**†	—	—	—	—	—	—	—	—	—
BRK	7	—	—	—	—	—	—	—	—	—	—
BVC	—	2**	—	—	—	—	—	—	—	—	—
BVS	—	2**	—	—	—	—	—	—	—	—	—
CLC	2	—	—	—	—	—	—	—	—	—	—
CLD	2	—	—	—	—	—	—	—	—	—	—
CLI	2	—	—	—	—	—	—	—	—	—	—
CLV	2	—	—	—	—	—	—	—	—	—	—

	Implied	Relative	Immediate	Zero page	Zero page, X	Absolute	Absolute, X	Absolute, Y	(Zero page, X)	(Zero page, Y)	(Indirect)
CMP	—	—	2	3	4	4	4*	4*	6	5*	5†
CPX	—	—	2	3	—	4	—	—	—	—	—
CPY	—	—	2	3	—	4	—	—	—	—	—
DEC	2†	—	—	5	6	6	7	—	—	—	—
DEX	2	—	—	—	—	—	—	—	—	—	—
DEY	2	—	—	—	—	—	—	—	—	—	—
EOR	—	—	2	3	4	4	4*	4*	6	5*	5†
INC	2†	—	—	5	6	6	7	—	—	—	—
INX	2	—	—	—	—	—	—	—	—	—	—
INY	2	—	—	—	—	—	—	—	—	—	—
JMP	—	—	—	—	6†	3	—	—	—	—	5
JSR	—	—	—	—	—	6	—	—	—	—	—
LDA	—	—	2	3	4	4	4*	4*	6	5*	5†
LDX	—	—	2	3	4	4	—	4*	—	—	—
LDY	—	—	2	3	4	4	4*	—	—	—	—
LSR	2	—	—	5	6	6	7	—	—	—	—
NOP	2	—	—	—	—	—	—	—	—	—	—
ORA	—	—	2	3	4	4	4*	4*	6	5	5†
PHA	3	—	—	—	—	—	—	—	—	—	—
PHX	3†	—	—	—	—	—	—	—	—	—	—
PHY	3†	—	—	—	—	—	—	—	—	—	—
PHP	3	—	—	—	—	—	—	—	—	—	—
PLA	4	—	—	—	—	—	—	—	—	—	—
PLX	4†	—	—	—	—	—	—	—	—	—	—
PLY	4†	—	—	—	—	—	—	—	—	—	—
PLP	4	—	—	—	—	—	—	—	—	—	—
ROL	2	—	—	5	6	6	7	—	—	—	—
ROR	2	—	—	5	6	6	7	—	—	—	—

	Implied	Relative	Immediate	Zero page	Zero page, X	Absolute	Absolute, X	Absolute, Y	(Zero page, X)	(Zero page, Y)	(Indirect)
RTI	6	—	—	—	—	—	—	—	—	—	—
RTS	6	—	—	—	—	—	—	—	—	—	—
SBC	—	—	2	3	4	4	4*	4*	6	5*	5†
SEC	2	—	—	—	—	—	—	—	—	—	—
SED	2	—	—	—	—	—	—	—	—	—	—
SEI	2	—	—	—	—	—	—	—	—	—	—
STA	—	—	—	3	4	4	5	5	6	6	5
STX	—	—	—	3	4	4	—	—	—	—	—
STY	—	—	—	3	4	4	—	—	—	—	—
STZ	—	—	—	3†	4†	4†	5†	—	—	—	—
TAX	2	—	—	—	—	—	—	—	—	—	—
TAY	2	—	—	—	—	—	—	—	—	—	—
TRB	—	—	—	5†	—	6†	—	—	—	—	—
TSB	—	—	—	5†	—	6†	—	—	—	—	—
TSX	2	—	—	—	—	—	—	—	—	—	—
TXA	2	—	—	—	—	—	—	—	—	—	—
TXS	2	—	—	—	—	—	—	—	—	—	—
TYA	2	—	—	—	—	—	—	—	—	—	—

*Add 1 cycle if page boundary crossed.
**Add 1 if branch occurs to same page or add 2 if branch occurs to a different page.
†65C02 only

5 Apple // Memory Map



6 Branch Calculators

The branch calculators are used to give branch values in hex. First, count the number of bytes you need to branch. Then locate this number in tators are used to give branch values in hex. First, count the number of bytes you need to branch. Then locate this number in the center of the appropriate table, and finally, read off the high and low hex nibbles from the side column and top row respectively.

Example: For a backward branch of 16 bytes:

Locate 16 in the center of Table A6.1 (bottom row),then read off high nibble (#F) and low nibble (#0) to give displacement value (#F0).

Table A6.1 Backward branch calculator

LSD / MSD	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
8	128	127	126	125	124	123	122	121	120	119	118	117	116	115	114	113
9	112	111	110	109	108	107	106	105	104	103	102	101	100	99	98	97
A	96	95	94	93	92	91	90	89	88	87	86	85	84	83	82	81
B	80	79	78	77	76	75	74	73	72	71	70	69	68	67	66	65
C	64	63	62	61	60	59	58	57	56	55	54	53	52	51	50	49
D	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33
E	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17
F	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1

Table A6.2 Forward branch calculator

LSD / MSD	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
2	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
3	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
4	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
5	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
6	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
7	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127

7 6502 and 65C02

All numbers are hexadecimal.

00 BRK implied
01 ORA (zero page, X)
02 Future expansion
03 Future expansion
*04 TSB zero page
05 ORA zero page
06 ASL zero page
07 Future expansion
08 PHP implied
09 ORA #immediate
0A ASL accumulator
0B Future expansion
*0C TSB absolute
0D ORA absolute
0E ASL absolute
0F Future expansion
10 BPL relative
11 ORA (zero page), Y
*12 ORA (indirect)
13 Future expansion
*14 TRB zero page
15 ORA zero page, X
16 ASL zero page, X
17 Future expansion
18 CLC implied
19 ORA absolute, Y
*1A INC implied
1B Future expansion
*1C TRB absolute
1D ORA absolute, X
1E ASL absolute, X
1F Future expansion
20 JSR absolute
21 AND (zero page, X)
22 Future expansion
23 Future expansion
24 BIT zero page
25 AND zero page
26 ROL zero page
27 Future expansion
28 PLP implied
29 AND #immediate
2A ROL accumulator
2B Future expansion
2C BIT absolute
2D AND absolute
2E ROL absolute
2F Future expansion
30 BMI relative
31 AND (zero page), Y
32 Future expansion
33 Future expansion
*34 BIT zero page, X
35 AND zero page, X
36 ROL zero page, X
37 Future expansion

38 SEC implied
39 AND absolute, Y
*3A DEC implied
3B Future expansion
*3C BIT absolute, X
3D AND absolute, X
3E ROL absolute, X
3F Future expansion
40 RTI implied
41 EOR (zero page, X)
42 Future expansion
43 Future expansion
44 Future expansion
45 EOR zero page
46 LSR zero page
47 Future expansion
48 PHA implied
49 EOR #immediate
4A LSR accumulator
4B Future expansion
4C JMP absolute
4D EOR absolute
4E LSR absolute
4F Future expansion
50 BVC relative
51 EOR (zero page), Y
*52 EOR (indirect)
53 Future expansion
54 Future expansion
55 EOR zero page, X
56 LSR zero page, X
57 Future expansion
58 CLI implied
59 EOR absolute, Y
*5A PHY implied
5B Future expansion
5C Future expansion
5D EOR absolute, X
5E LSR absolute, X
5F Future expansion
60 RTS implied
61 ADC (zero page, X)
62 Future expansion
63 Future expansion
*64 STZ zero page
65 ADC zero page
66 ROR zero page
67 Future expansion
68 PLA implied
69 ADC #immediate
6A ROR accumulator
6B Future expansion
6C JMP (indirect)
6D ADC absolute
6E ROR absolute
6F Future expansion
70 BVS relative
71 ADC (zero page), Y
72 Future expansion
73 Future expansion
*74 STZ (zero page, X)
75 ADC zero page, X
76 ROR zero page, X
77 Future expansion
78 SEI implied
79 ADC absolute, Y
*7A PLY implied
7B Future expansion
*7C JMP (zero page, X)
7D ADC absolute, X
7E ROR absolute, X
7F Future expansion
*80 BRA relative
81 STA (zero page, X)
82 Future expansion
83 Future expansion
84 STY zero page
85 STA zero page

86 STX zero page
87 Future expansion
88 DEY implied
*89 BIT immediate
8A TXA implied
8B Future expansion
8C STY absolute
8D STA absolute
8E STX absolute
8F Future expansion
90 BCC relative
91 STA (zero page), Y
*92 STA (indirect)
93 Future expansion
94 STY zero page, X
95 STA zero page, X
96 STX zero page, Y
97 Future expansion
98 TYA implied
99 STA absolute, Y
9A TXS implied
9B Future expansion
*9C STZ absolute
9D STA absolute, X
*9E STZ absolute, X
9F Future expansion
A0 LDY #immediate
A1 LDA (zero page, X)
A2 LDX #immediate
A3 Future expansion
A4 LDY zero page
A5 LDA zero page
A6 LDX zero page
A7 Future expansion
A8 TAY implied
A9 LDA #immediate
AA TAX implied
AB Future expansion
AC LDY absolute
AD LDA absolute
AE LDX absolute
AF Future expansion
B0 BCS relative
B1 LDA (zero page), Y
*B2 LDA (indirect)
B3 Future expansion
B4 LDY zero page, X
B5 LDA zero page, X
B6 LDX zero page, Y
B7 Future expansion
B8 CLV implied
B9 LDA absolute, Y
BA TSX implied
BB Future expansion
BC LDY absolute, X
BD LDA absolute, X
BE LDX absolute, Y
BF Future expansion
C0 CPY #immediate
C1 CMP (zero page, X)
C2 Future expansion
C3 Future expansion
C4 CPY zero page
C5 CMP zero page
C6 DEC zero page
C7 Future expansion
C8 INY implied
C9 CMP #immediate
CA DEX implied
CB Future expansion
CC CPY absolute
CD CMP absolute
CE DEC absolute
CF Future expansion
D0 BNE relative
D1 CMP (zero page), Y
*D2 CMP (indirect)
D3 Future expansion

D4 Future expansion
D5 CMP zero page, X
D6 DEC zero page, X
D7 Future expansion
D8 CLD implied
D9 CMP absolute, Y
*DA PHX implied
DB Future expansion
DC Future expansion
DD CMP absolute, X
DE DEC absolute, X
DF Future expansion
E0 CPX #immediate
E1 SBC (zero page, X)
E2 Future expansion
E3 Future expansion
E4 CPX zero page
E5 SBC zero page
E6 INC zero page
E7 Future expansion
E8 INX implied
E9 SBC #immediate
EA NOP implied
EB Future expansion
E0 CPX #immediate
E1 SBC (zero page, X)
E2 Future expansion
E3 Future expansion

E4 CPX zero page
E5 SBC zero page
E6 INC zero page
E7 Future expansion
E8 INX implied
E9 SBC #immediate
EA NOP implied
EB Future expansion
EC CPX absolute
ED SBC absolute
EE INC absolute
EF Future expansion
F0 BEQ relative
F1 SBC (zero page), Y
*F2 SBC (indirect)
F3 Future expansion
F4 Future expansion
F5 SBC zero page, X
F6 INC zero page, X
F7 Future expansion
F8 SED implied
F9 SBC absolute, Y
*FA PLX implied
FB Future expansion
FC Future expansion
FD SBC absolute, X
FE INC absolute, X
FF Future expansion

*65C02 op code

General Index

Program Index